DEEP LEARNING TECHNIQUES FOR TEXT CLASSIFICATION

DIARDANO RAIHAN

SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING

2021

DEEP LEARNING TECHNIQUES FOR TEXT CLASSIFICATION

DIARDANO RAIHAN

SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING

A DISSERTATION SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN COMPUTER CONTROL AND AUTOMATION

2021

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

9 April 2021

Date

Diardano Raihan

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

9 April 2021

Date

Prof. P. N. Suganthan

Authorship Attribution Statement

This thesis **does not** contain any materials from papers published in peer-reviewed journals or from papers accepted at conferences in which I am listed as an author.

9 April 2021

Date

Diardano Raihan

Table of Contents

Abstract

This dissertation presents a series of experiments in applying deep learning techniques for text classification. The experiment will evaluate the performance of some popular deep learning models, such as feedforward, recurrent, convolutional, and ensemble-based neural networks, on five different datasets. We will build each model on top of two separate feature extractions to capture information within the text. The result shows that the word embedding provides a robust feature extractor to all the models in making a better final prediction. The experiment also highlights the effectiveness of the ensemble-based and temporal convolutional neural network in achieving good performances and even competing with the state-of-the-art benchmark models.

Acknowledgements

Firstly, I would like to express my gratitude towards my supervisor, Assoc. Prof. P. N. **Suganthan**, for the opportunity to work in an impressive area such as Deep Learning and his guidance, regular supervision, constructive feedback, and suggestions in completing the project.

Secondly, I would like to express my special gratitude to the assistant of my supervisor, Mr. **Cheng Wen Xin**, for his attention, time, and assistance during this work. Without his initiative and sincere help, this dissertation would not have been possible in time.

Thirdly, I would like to acknowledge with much appreciation the crucial role of my sponsor, the Indonesia Endowment Fund for Education (**LPDP** - Lembaga Pengelola Dana Pendidikan). Thanks to generous support, I can pursue my passion for a master of science in the field that I love the most and at one of the best universities in the world. I hope the knowledge and networking I gain could bring benefit to Indonesia soon.

Lastly, but most importantly, I wish to thank my parents, my beloved little brother, and my friends for their love and continuous support throughout my life. Thank you for always giving me prayers, strength, and faith to successfully finish this dissertation and keep motivating me to chase my dreams.

Acronyms

1D	One-Dimensional
AI	Artificial Intelligence
AVG	Average
ARV	Average Rank Values
BiGRU/BiLSTM	Bidirectional Gated Recurrent Unit / Long Short-Term Memory
BoF	Bag-of-Features
BoW	Bag-of-Words
CBOW	Continuous Bag-of-Words
CNN	Convolutional Neural Network
CR	Customer Review
CV	Cross-Validation
edRVFL	Ensemble Deep Random Vector Functional Link
GloVe	Global Vector
GRU	Gated Recurrent Unit
LSA	Latent Semantic Analysis
LSTM	Long Short-Term Memory
MPQA	Multi-Perspective Question Answering
MR	Movie Review
NLP	Natural language Processing
NTU	Nanyang Technological University
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Networks
RMDL	Random Multimodel Deep Learning
RVFL	Random Vector Functional Link
SELU	Scaled Exponential Linear Unit
SGNS	Skip-Gram with Negative Sampling
SNN	Shallow Neural Network
SOTA	State-of-the-Art
SUBJ	Subjectivity
TCN	Temporal Convolutional Network
TF-IDF	Term Frequency-Inverse Document Frequency
TREC	Text REtrieval Conference
Word2Vec	Word to Vector
WE	Word Embedding

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the concept of Deep Learning techniques in a subfield of Artificial Intelligence (AI), namely Natural language Processing (NLP) for a text classification task. We will discuss the project's motivation, the objectives, the scope, and how we organize it.

1.1 Motivation

Natural Language Processing (NLP) is automating or manipulating natural human language (text and speech) by a program. NLP has many essential tasks, such as text classification, image captioning, language modeling, machine translation, and many more.

Text classification is one of the popular tasks in NLP that allows a program to classify free-text documents based on pre-defined classes. The classes can be based on topic, genre, or sentiment. As an illustration, the followings are applications commonly used nowadays:

Table 1.1 Applications of the text classification task.

Text Classification Applications	Illustration
Sentiment Analysis	Identifying a product feedback either having positive, negative, or neutral sentiment.
Information Filtering	Selecting relevant information from a stream of text data.
Information Retrieval	Locating documents that comply with the information needed within extensive collections of documents.
Recommender System	Suggesting products to users based on their description and the user's interests.
Document Categorization	Detecting unwanted email as spam, categorizing a news article.

Today's emergence of large digital documents makes the text classification task more crucial, especially for companies to maximize their workflow or even profits. Hence, many fields have used this task globally, including engineering, medicine, social science, healthcare, law, and many others.

Recently, the progress of NLP research on text classification has arrived at the state-of-the-art (SOTA). It has achieved terrific results, showing Deep Learning methods as the cutting-edge technology to perform such tasks. Some exciting progress is as follows:

1. In 2014, Yoon Kim [1] figured out that a shallow Convolutional Neural Network (CNN) with word embedding works well despite little tuning of hyperparameters.
2. In 2018, Shaojie Bai et al. [2] introduced a generic temporal convolutional network (TCN) that can be applied for all tasks. The paper proved the model outperforms recurrent architecture in the sequence modeling task.
3. In 2018, Kowsari, K. et al. [3] used an ensemble method to combine multiple deep learning algorithms into a single big model called Random Multimodel Deep Learning (RMDL) on the text classification task. The model offers a solution to find the best deep learning structure and architecture.
4. In 2019, Rakesh Katuwal et al. [4] proposed an ensemble deep Random Vector Functional Link (edRVFL) network that shows outstanding performance on various domains.
5. In 2020, Beakcheol Jang et al. [5] built a hybrid deep learning model by combining Bidirectional Long Short-Term Memory (LSTM) with CNN and word embedding to increase accuracy in text classification.

As mentioned above, TCN has the potential to make recurrent architecture "out-of-date". On the other hand, ensemble-based models tend to have better performance than a single-based. As such, it is always appealing to find a proper way to build the best model and configuration to perform a specific task. Hence, the need to assess the performance of the SOTA deep learning models for text classification is essential not only for academic purposes but also for AI practitioners or professionals that need guidance and benchmark on similar projects.

1.2 Objectives and Scope

We will perform experiments on the performance of deep learning techniques on text classification datasets. The deep learning model on each experiment is based on one central architecture evaluated on five text classification datasets as follows:

Table 1.2 The scope of the dissertation.

Deep Learning Models	Feedforward, CNN, RNN, Ensemble-learning models.
Datasets	Subjectivity, Movie Review, Customer Review, Text Retrieval Conference, Multi-Perspective Question Answering
Feature Extractions	Bag-of-Words, Word Embedding, Word2Vec.

To ease the comparison for model performances, we will use accuracy as the only metric.

1.3 Major Contribution of the Dissertation

The dissertation gives insight on how to build the best deep learning model for text classification. We compare each model accuracy with the benchmark of SOTA models. The comparison allows us to provide some recommendations on picking the architectures, tuning the hyperparameters, and choosing what feature extraction works best for each dataset. In the end, we also propose the best starting model to build for future applications.

1.4 Organization of the Dissertation

We organize the dissertation into six chapters. Chapter 1 discusses the background and motivation that lead to the objective and scope of the project. Next, Chapter 2 presents the literature review of standard feature extractions and popular deep learning models. Then, Chapter 3 introduces the models and hyperparameters used for the experiment. Chapter 4 focuses on the datasets, data pre-processing, and experimental setup. Subsequently, Chapter 5 will present the results and evaluation for each model compared to the benchmarks. Finally, Chapter 6 consists of the project summary, conclusion, and recommendation for future work. We also include the implementation codes in the Appendix section.

Chapter 2

Literature Review

This chapter reviews common feature extractions for text data and deep learning architectures that we will use in the experiment.

2. 1 Feature Extractions

Text is raw unstructured data that we cannot easily feed it directly into deep learning models. It needs special preparation to represent the information within. Text data should be encoded as numbers, precisely a vector of numbers. Thus, the process is called feature extraction. Then, we can use the vector as input or output for the model. This chapter will discuss two standard feature extraction techniques that we will use in this project.

2.1.1 Bag-of-Words (BoW)

The bag-of-word (BoW) is a simple and easy method to represent text as the number of word occurrences within a document. As the name implies, BoW converts a sentence or document into a bag of words taking the form of a list. Hence, the method discards the order of words in the document. As a result, the model will learn whether or not the words appear in the document.

The BoW has four standard scoring word options as follows:

- **Binary**. It marks the word present in the current document with a Boolean value (0/1).
- **Count**. It counts the word occurrences in the current document with an integer value.
- **Freq**. It calculates the word occurrences over all words in the current document.
- **TF-IDF**. It refers to the Term Frequency-Inverse Document Frequency scoring for each word in the document.

2.1.1.1 Term Frequency-Inverse Document Frequency (TF-IDF)

The Term Frequency-Inverse Document Frequency refers to the BoW scoring method by giving penalty of frequent words in the corpus that start to dominate. The familiar words will be problematic if they do not have much informational content to the model. Hence, K. Sparck Jones [6] offered a method called Inverse Document Frequency (IDF) to ease this common word's effect. The method will score how infrequent the word is within all documents. IDF is often used with Term Frequency (TF) to score the word occurrence in the current document. The TF-IDF formula is given in Equation (1):

$$W_{i,j} = tf_{i,j} \times \log_{10}\left(\frac{N}{df_i}\right) \tag{1}$$

- $tf_{i,j}$: the number of times word *i* appears in *j* over the total number of terms in *j*.
- df_i : the number of documents having *i*.
- N : the total number of documents.

The first part of Equation (1) contains each term (word) percentage in the given document. The second part calculates how often this term (word) appears across all the documents. Thus, the rarer the word is, the higher the value will be. In other words, the TF-IDF helps pull out prominent but rarely-used words.

Here is an illustration of BoW:

- Document

"I have a great experience at NTU."
"NTU is a world-class university."
"I love NTU and all the people in the campus."
"The quality of education at NTU is one of the best in the world."

- Bag-of-Words (BoW)

[*"the", "ntu", "i", "a", "at", "is", "world", "in", "of", "have", "great", "experience", "class", "university", "love", "and", "all", "people", "campus", "quality", "education", "one", "best"*]

- Bag-of-Feature (BoF) using **binary** scoring

```
[0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0]
[0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
```

- Bag-of-Feature (BoF) using **count** scoring

```
[0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 2, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0]
[0, 3, 1, 0, 0, 1, 1, 1, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
```

- Bag-of-Feature (BoF) using **freq** scoring

```
[0, 0, 0.14, 0.14, 0.14, 0.14, 0, 0, 0, 0, 0.14, 0.14, 0.14, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0.17, 0, 0.17, 0, 0.17, 0.17, 0, 0, 0, 0, 0, 0.17, 0.17, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0.2, 0.1, 0.1, 0, 0, 0, 0, 0.1, 0, 0, 0, 0, 0, 0, 0.1, 0.1, 0.1, 0.1, 0.1, 0, 0, 0, 0]
[0, 0.21, 0.07, 0, 0, 0.07, 0.07, 0.07, 0.07, 0.14, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.07, 0.07, 0.07, 0.07]
```

- Bag-of-Feature (BoF) using **TF-IDF** scoring

```
[0, 0, 0.59, 0.85, 0.85, 0.85, 0, 0, 0, 0, 1.1, 1.1, 1.1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0.59, 0, 0.85, 0, 0.85, 0.85, 0, 0, 0, 0, 0, 1.1, 1.1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1.43, 0.59, 0.85, 0, 0, 0, 0, 0.85, 0, 0, 0, 0, 0, 0, 1.1, 1.1, 1.1, 1.1, 1.1, 0, 0, 0, 0]
[0, 1.78, 0.59, 0, 0, 0.85, 0.85, 0.85, 0.85, 1.86, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1.1, 1.1, 1.1, 1.1]
```

2.1.1.2 Advantages of BoW

As we can see, the BoW is easy to understand and implement. The intuition here is that documents will be similar for similar contents. It also gives us the flexibility to customize the encoding process. It is pretty effective for document classification in a small corpus.

2.1.1.3 Disadvantages of BoW

Despite the simplicity the BoW can offer, it causes several problems such as:

- **Vocabulary**. We need to build vocabulary first before converting our text into numbers. Hence, if the document is large, then we will have a large size of the vocabulary. An extensive vocabulary can lead to sparsity.
- **Sparsity**. The sparse representation makes the model harder to train (computationally) and harness information from the input features.
- **Context and Meaning**. Since we discard the structure, we ignore the context and the semantics of the document. In NLP, context and meaning can provide a robust model capable of identifying synonyms, the same words differently ordered, and much more.

In the next section, we will discuss word embedding, a powerful method to represent text that can carry its meaning and context.

2.1.2 Word Embeddings (WE)

We have seen that the BoW fails to carry the context and meaning in a text or document. Word embeddings are a method to represent a word as a vector that carries meaning. It allows words with similar meanings to have almost equal representation. For example, the word *"happy"* and *"excited"* are often used in the same context. Thus, those words will have similar vector representations. The vector size of word embedding will relatively have a low dimension ranging between 10 - 300. The vector makes the representation practical for calculations. Since each vector carries the word meaning, it is now possible to find the analogy and specify how close the words are semantically. Figure 2.1 illustrates the use of word embedding to represent any word in one and two-dimensional vectors.

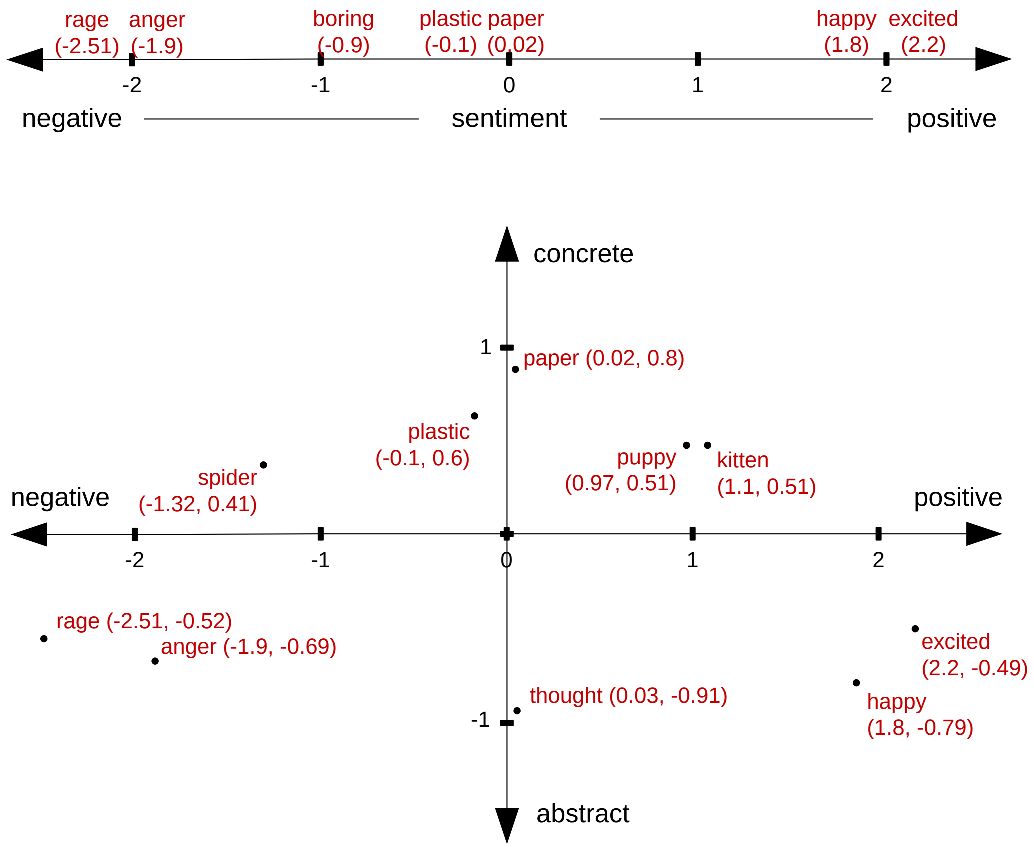


Figure 2.1 The illustration of word embedding in one and two-dimensional vectors.

2.1.2.1 Embedding Layer

An embedding layer is a word embedding method stacked in front of the main neural network model. The layer consists of parameters trained jointly with the model on a specific NLP task, such as text classification. That is, those parameters are the vectors. Firstly, we initialize the vectors with small random numbers. Then, while the model is training, we also update the word embedding parameters using the backpropagation algorithm. In the end, each word will be mapped into an N-dimensional vector, where N is the size of the word embedding vector.

Nowadays, many researchers have built well-established word embedding methods for machine learning models. Generally, the methods use a self-supervised (unsupervised + supervised) learning task using document statistics. It is unsupervised because the text data is unlabeled. However, it is supervised because the data provides sufficient context, which will typically make up the labels.

Among many word embedding methods, the most prominent ones are Word2Vec by Google [7], GloVe by Stanford University [23], FastText by Facebook [24]. These word embedding methods have been proven to work well in deep learning applications. We will focus on the Word2Vec as the pre-trained word embedding in this dissertation.

2.1.2.2 Word2Vec

In 2013, T. Mikolov et al. [7] at Google proposed a method called Word2Vec to learn a stand-alone word embedding efficiently from a text corpus. The Word2Vec uses a shallow neural network to generate an N-dimensional vector for each word. It proposes two model architectures:

- the Continuous Bag-of-Words (CBOW);
- the Continuous Skip-gram / Skip-gram with Negative Sampling (SGNS).

The CBOW and Skip-gram models are powerful tools to discover the relationships and similarities between words. They can keep the semantic and syntactic information of text data for machine learning algorithms. As illustrated in Figure 2.2 [7], the CBOW model predicts the missing word based on the context (surrounding words). On the other hand, the Skip-gram model is the reverse of CBOW, predicting the context based on the current word [7].

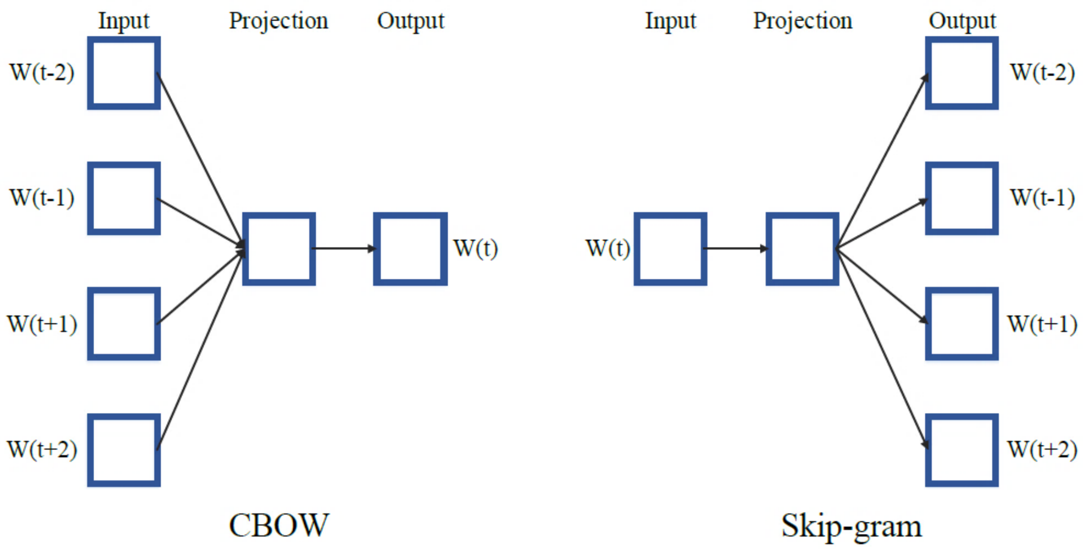


Figure 2.2 The CBOW and the Skip-gram architectures.

The number of neurons in the hidden layer represents the size of expected word embedding, N. Then, the parameters are learned and extracted either from the weights between the input and hidden layer, hidden and output layer, or both combined and averaged. These parameters then what we call the pre-trained word embedding - Word2Vec. The limitation of Word2Vec is that the model cannot provide word embeddings for words the models did not see in the corpus during the training process.

Google has published a pre-trained Word2Vec containing 3 million words and phrases with their respective vector representation. It has 300-dimensional word vectors and has been trained on Google news (100 billion words). The file size is around 1.53 GB.

2.1.2.3 Global Vectors for Word Representation (GloVe)

The Global Vector for Word Representation (GloVe) [23] is another popular word embedding method that has been commonly used for NLP applications, such as text classification. In 2014 at Stanford, Pennington et al. developed a method to create a vector text representation without even using a neural network at all, such as in Word2Vec. Instead, the method involves factorizing a co-occurrence matrix such as Latent Semantic Analysis (LSA). This method will factorize the logarithm of corpus words in the co-occurrence matrix and use global text statistics to capture the meaning and analogies.

The GloVe does not use a sliding window to capture the local context. Instead, it will build a word co-occurrence matrix or a clear word context using the whole text corpus statistics. This method combines the concept of local context in Word2Vec with global statistics like LSA.

The following is the GloVe objective function:

$$f\left(w_i - w_j, \widetilde{w_k}\right) = \frac{P_{ik}}{P_{jk}} \quad (2)$$

- w_i : the word vector of word i.
- P_{ik} : the probability of word k to appear in the context of word i.

Like Word2Vec, Stanford also has published a pre-trained version of GloVe. The downloaded pre-trained GloVe comes with four different models based on vector dimensions (50, 100, 200, and 300). The smallest version of GloVe is trained on

Wikipedia, containing 6 billion words and phrases with 400,000 vocabularies. The training process results in a file size of around 822 MB.

2.1.2.4 FastText

In 2016, Facebook AI Research Lab created FastText [24] as a novel word embedding technique based on the skip-gram model. The FastText considers the word structure by representing each word as a bag-of-character, N-gram of characters. The motivation behind FastText is that many word embedding methods neglect the word morphology and directly assign each word to a different vector [24]. As an illustration, given the word *"technology"*, with $n = 3$, FastText will generate the tri-grams representation of the word as follows:

$$< te, tec, ech, chn, hno, nol, olo, log, ogy, gy >$$

Suppose G is the size of an n-grams dictionary, z_g is the vector representation of a given word w to each gram g, then the scoring function [24] for this is obtained by the following formula:

$$s(w, c) = \sum_{g \in g_w} z_g^T v_c \tag{3}$$

where $g_w \in \{1,2, \dots, G\}$

As a result, FastText supports any word that is not in their vocabulary list because of having different word morphology. For example, FastText will provide a similar vector representation for the word "*kitty*" and "*kitten*", although it has yet to see *"kitten"* before.

Like Word2Vec, Facebook has also published a pre-trained version of FastText with 300-dimensional word vector representation. The pre-trained FastText are trained on Wikipedia and available in 294 languages.

2. 2 Deep Learning Models

Deep Learning is a subfield of Machine Learning techniques that uses a layered representation of data known as Neural Networks. The word "deep" indicates a large or deep neural network. Recently, deep learning models have reached the SOTA results for many applications, including text and document classification. We will review some basic and advanced architectures used in this dissertation.

2.2.1 Feedforward Neural Networks

The feedforward neural network is the basic architecture that consists of multiconnection of layers straight forward from input to output layer. Hence, each layer only receives information from the previous one and carries on the information to the next layer until it reaches the output. The input layer represents the essential feature extracted from the data. In the text classification, the input can be constructed via BoW or word embedding. The output corresponds to the number of classes the model tries to classify. Figure 2.3 [25] illustrates the standard structure of the feedforward deep neural network.

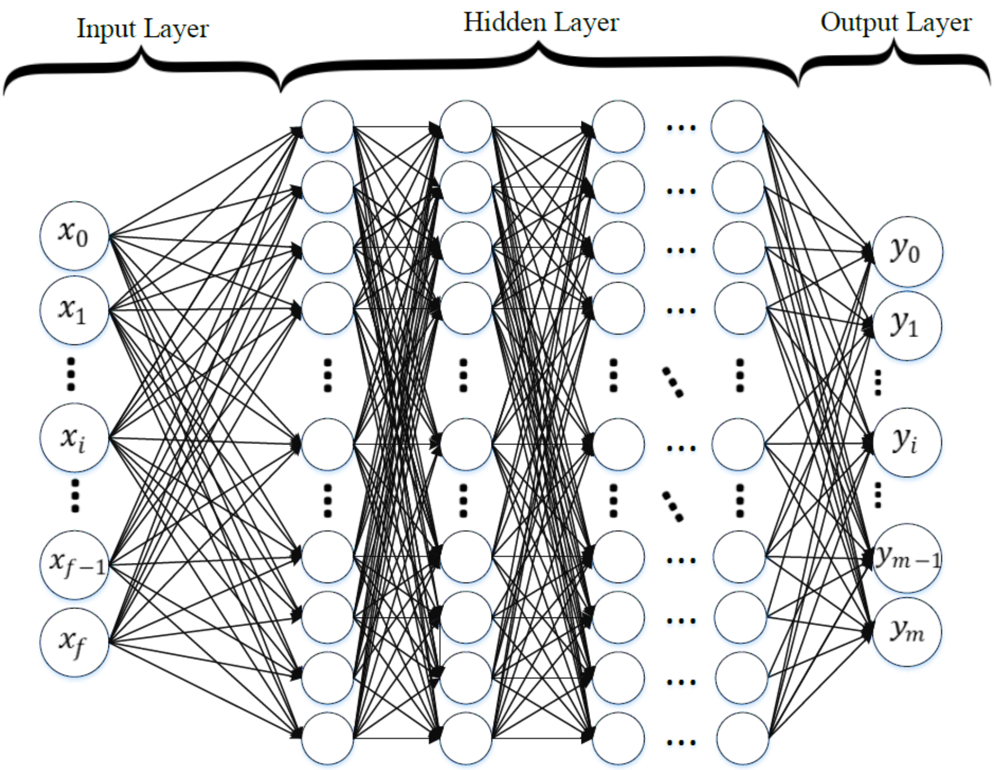


Figure 2.3 The standard architecture of a fully connected deep neural network.

Given a set of input and output (target), the model will learn the connection between using hidden layers. The standard activation function in the hidden units is to use sigmoid (Equation (4)) or Rectified Linear Unit (ReLU) (Equation (5)). The output activation function uses sigmoid for binary classification or softmax (Equation (6)) for multi-class classification. The model is trained using a standard backpropagation algorithm.

$$f(z) = \frac{1}{1+e^{-z}} \in (0,1) \tag{4}$$

$$f(z) = \max(0, z) \tag{5}$$

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}} \forall\, i \in \{1, \dots, K\} \tag{6}$$

2.2.1.1 Random Vector Functional Link (RVFL)

The Random Vector Functional Link presents as an alternative feedforward neural network architecture for any deep learning tasks. The basic RVFL is motivated by the presence of backpropagation weaknesses, such as multiple local minimums, slow convergence because of weight adjustments on every link, and learning rate sensitivity [13] that generally occurred in any feedforward model. It provides an alternative for the backpropagation algorithm where parameters (weights β and biases b) from input to hidden layer H are randomly generated in the proper range and kept fixed (blue lines in Figure 2.4 [12]). The values in hidden layers are calculated using Equation (7):

$$H = g(X\beta + b) \tag{7}$$

where X is the input features and $g(z)$ is the activation function.

At the output layer, the input X and hidden layers H are concatenated such that $D = [H, X]$ to feed the output neurons. In the end, we only need to compute the output weights β_s using a closed-form solution. The closed-form solution using the regularized least square is given below:

$$\text{Primal space:} \qquad \beta_s = (\lambda I + D^T D)^{-1} D^T Y \tag{8}$$

$$\text{Dual space:} \qquad \beta_s = D^T (\lambda I + D D^T)^{-1} Y \tag{9}$$

where λ is the regularization parameter, and Y is the target vector.

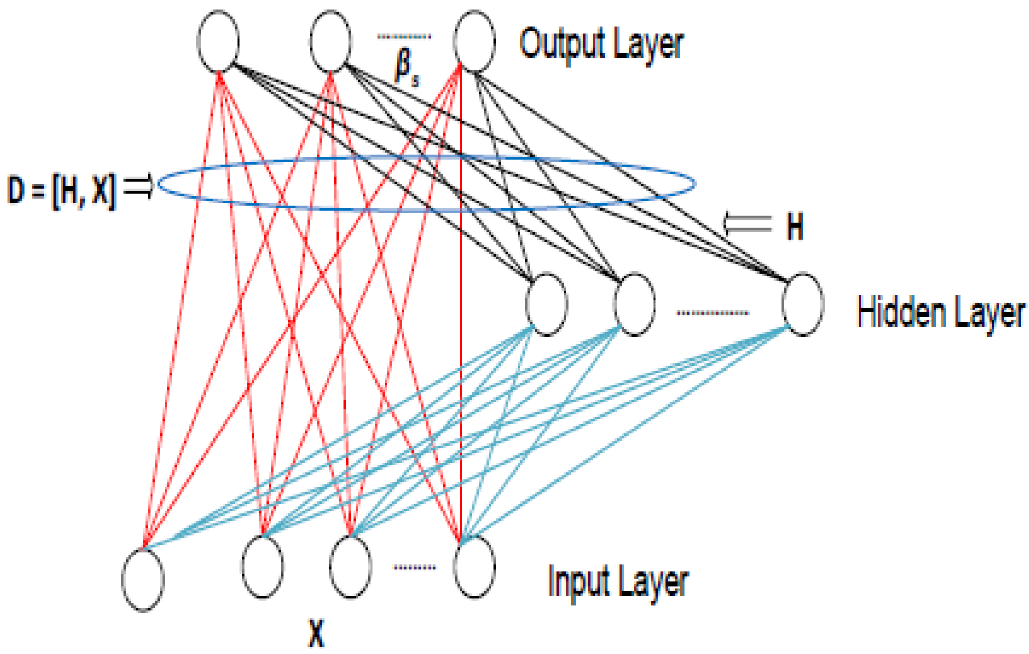


Figure 2.4 The basic structure of an RVFL network.

2.2.2 Recurrent Neural Networks (RNN)

The recurrent neural network allows the information to be transferred in a sophisticated way, where it scans the data in specific time steps from left to the right. Hence, the parameters are shared for each time step and updated. When the model makes a prediction, not only does it get the information from the current input x_t but also from $x_{t-1}$and x_{t-2}. This is because the previous information can pass through the model to help the current prediction. The formula is illustrated below:

$$a_t = f(W_{rec}a_{t-1} + W_{in}x_t + b) \tag{10}$$

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{11}$$

where a_t is the current state in the hidden layer and x_t is the current input at time t. W_{rec} is the shared recurrent parameters, W_{in} is the input parameters, and b is the bias. The standard activation function for RNN is the hyperbolic tangent function or tanh as in Equation (11).

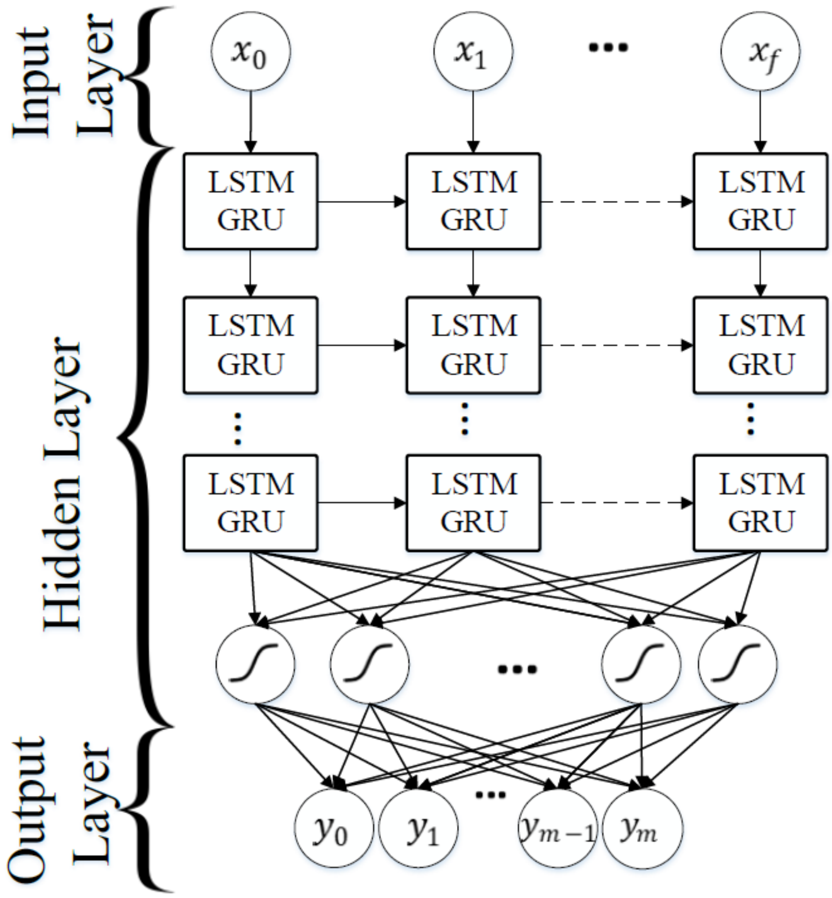


Figure 2.5 The standard LSTM/GRU recurrent neural networks.

However, the standard RNN generally suffers from vanishing and exploding gradient problems when the gradient descent error is propagated back to the network [8]. Thus, we mostly work with LSTM or GRU as the more advanced models, as shown in Figure 2.5 [25]. Next, RNN only uses the previous information to make a prediction. In many cases, we can find the context where we see the future information, not just the past, such as:

- **He said**, *"**Teddy** Roosevelt was a great leader".*
- **He said**, *"**Teddy** bears are really cute".*

Clearly, we cannot know for sure if the word *"Teddy"* is a person's name given only the first three words. Therefore, the solution is to use the **Bidirectional RNN**, where it allows us to process information from both earlier and later in the sequence.

2.2.2.1 Gated Recurrent Unit (GRU)

The Gated Recurrent Unit (GRU) is the modification to the standard RNN model with a gating mechanism formulated by J. Chung et al. [9]. It contains two gates called reset gate, Γ_r and update gate, Γ_u. It uses the network hidden units as the memory cells called cell state, C_t, such that $h_t = C_t$, as in Figure 2.6 [25]. The cell state will supply a little memory to remember. The complete formula of GRU is as follows:

$$\widetilde{C_t} = \tanh\left(W_c[\Gamma_r \circ C_{t-1}, x_t] + b_c\right) \tag{12}$$

$$\Gamma_u = \sigma\left(W_u[C_{t-1}, x_t] + b_u\right) \tag{13}$$

$$\Gamma_r = \sigma\left(W_r[C_{t-1}, x_t] + b_r\right) \tag{14}$$

$$C_t = \Gamma_u \circ \widetilde{C_t} + (1 - \Gamma_u) \circ C_{t-1} \tag{15}$$

$$h_t = C_t \tag{16}$$

where Equation (12) is the candidate for replacing the current cell state in Equation (15), Equation (13) defines the update-gate, and Equation (14) is the reset-gate. The reset gate controls how relevant the previous cell state of computing the next cell state candidate. The activation function for each gate can be a sigmoid or ReLU. Hence, the gate values can be so close to zero, making the model not suffer from a vanishing gradient problem, causing $C_t = C_{t-1}$. The cell state value is maintained across many time steps (long sequence) until we need to use it.

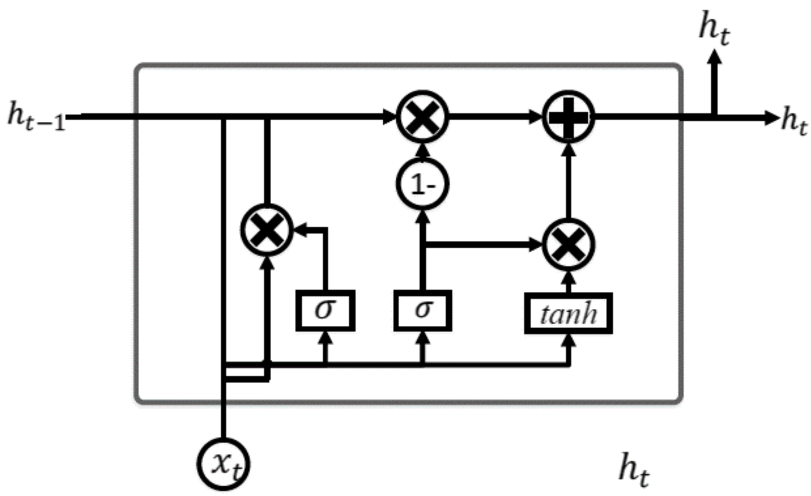


Figure 2.6 The Gated Recurrent Unit (GRU) cell.

2.2.2.2 Long Short-Term Memory (LSTM)

Introduced by S. Hochreiter and J. Schmidhuber [10], LSTM is a more complex model of GRU that can also learn a sequence with long-range connections. It has three gates: update gate Γ_u, forget gate Γ_f, and output gate Γ_o, make it more effective and powerful. Compared to GRU, LSTM has its internal memory cell or cell state, such that $h_t \neq C_t$. The graphical representation of LSTM is shown in Figure 2.7 [25] with the complete formula as follows:

$$\widetilde{C}_t = \tanh\left(W_c[h_{t-1}, x_t] + b_c\right) \tag{17}$$

$$\Gamma_u = \sigma\left(W_u[h_{t-1}, x_t] + b_u\right) \tag{18}$$

$$\Gamma_f = \sigma\left(W_f[h_{t-1}, x_t] + b_f\right) \tag{19}$$

$$\Gamma_o = \sigma\left(W_o[h_{t-1}, x_t] + b_o\right) \tag{20}$$

$$C_t = \Gamma_u \circ \widetilde{C}_t + \Gamma_f \circ C_{t-1} \tag{21}$$

$$h_t = \Gamma_o \tanh\left(C_t\right) \tag{22}$$

where Equation (18), (19), and (20) correspond to update, forget, and output gates, respectively. Equation (17) calculates the candidate memory cell value for updating the new one in Equation (21). Each gate and memory cell contains weights W and bias b, also input at time t and information at $t-1$. Although LSTM and GRU offer a promising result, the model can be biased when later words are more dominant than earlier ones. A convolutional neural network solves this issue by applying a max-pooling layer to specify discriminative phrases in text data [11].

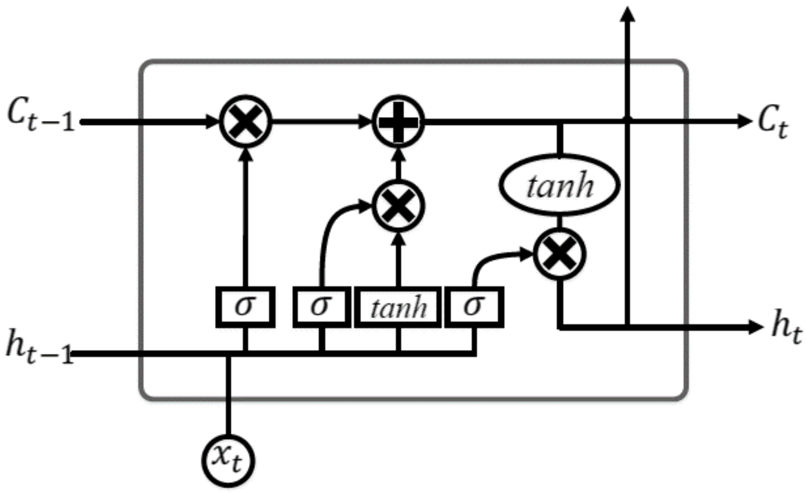


Figure 2.7 The Long Short-Term Memory (LSTM) cell.

2.2.2.3 Bidirectional RNN (BiRNN)

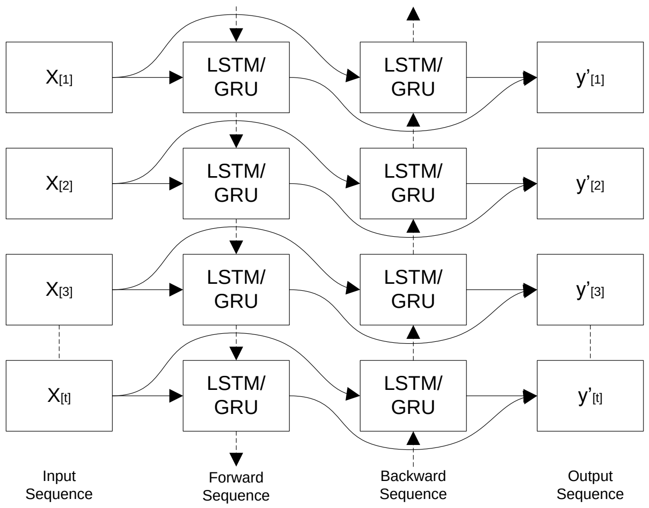


Figure 2.8 The bidirectional RNN.

Figure 2.8 shows the typical bidirectional configuration for the recurrent neural network. Given an input sequence $x_{[1]}$ to $x_{[t]}$, the RNN cells (LSTM/GRU) will operate in the forward and backward directions (or positive and negative time directions [26]). Operating in both directions will take past and future information into account to potentially predict from the entire sequence. The main disadvantage of BiRNN is that it requires the whole sequence of data before making any prediction. However, the disadvantage does not seem to be the case for a text classification task where typically, the whole sentence is already presented simultaneously.

2.2.3 Convolutional Neural Networks (CNN)

A convolutional neural network (CNN) is a deep learning architecture built initially for computer vision. Lately, CNN has shown to be effective in NLP applications, such as text classification [1]. CNN uses a set of filters (kernels or windows) of size d x d to perform convolution operations by passing it to local features. The output result is called a feature map. The feature map is followed by a pooling operation to compress the output size and reduce the computational complexity without eliminating essential features. The most pooling operation applied is max pooling, where it selects the maximum feature within a defined window.

After performing several convolution-pooling operations, the pre-final output will be flattened into one column followed by few fully connected layers to produce the final output. Figure 2.9 depicts the CNN architecture for text classification. The illustration contains word embedding to represent the text as the input layer, followed by 1D convolutional and pooling layers, fully connected layers, and output layer.

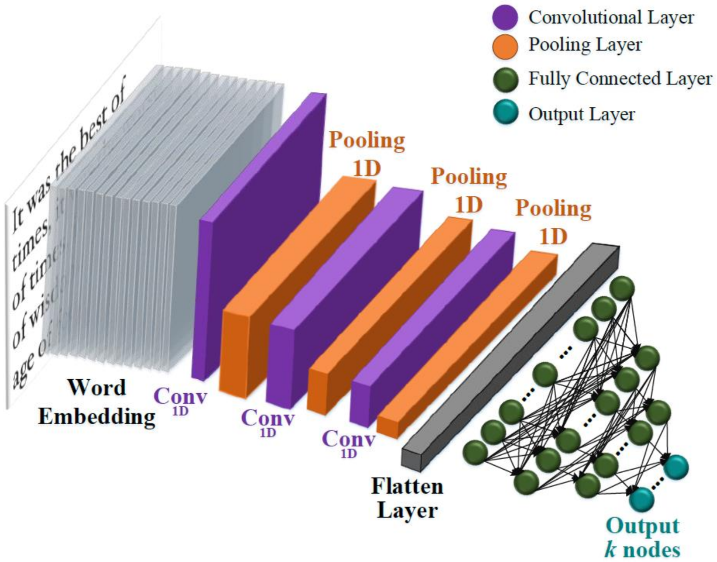


Figure 2.9 The CNN architecture for text classification.

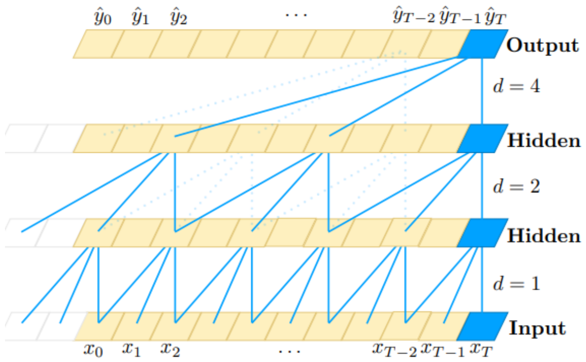
2.2.3.1 Temporal Convolutional Network (TCN)

Typically, the standard convolution in CNN will put the operation in the center of the filter, producing values in the past, present, and future. For some applications, it is inconvenient to consider values from the future, such as for time series applications, sequential sampling, and regression. Shaojie Bai et al. [2] proposed a generic temporal convolutional network (TCN) as a dilated-causal version of CNN to fix this issue.

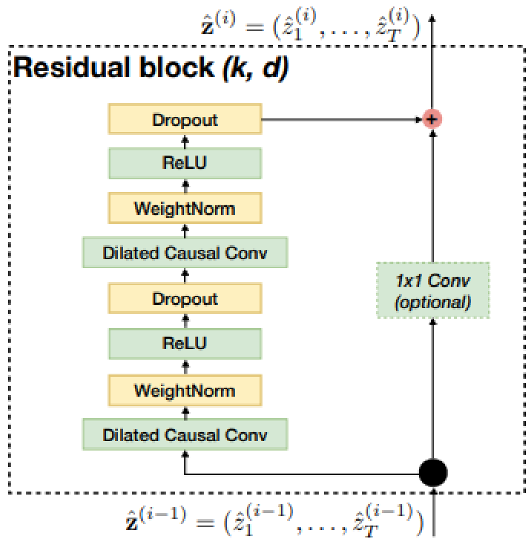
A TCN provides a causal convolution to produce the present value. Hence, the filter only applies to present and past inputs. For filter of size k, it will add padding of $k-1$ size at the beginning of sequence rather than both sides. The receptive field R is the number of elements that the convolutions can observe. To deal with long-range dependencies, a TCN dilates the filter by a dilation factor d such that the output at position t can rely on $d \times (k-1)$ steps of the past input values. Then, the convolutions will be stacked by different dilation factors to handle sparsity. The relationship between the receptive field R of a TCN with l layers and filter of size k is given in Equation (23).

$$R = 2^l(k-1) \tag{23}$$

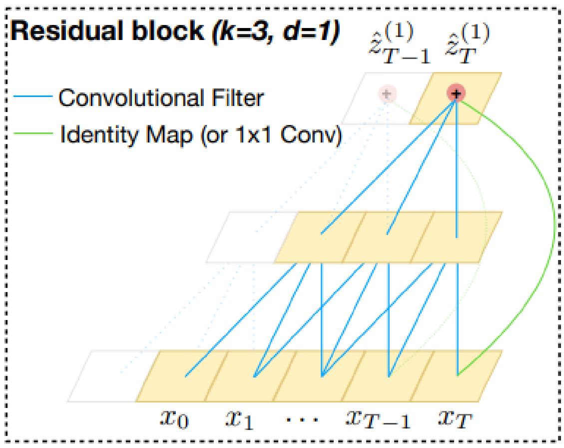
To prevent the vanishing gradient problems, a TCN uses batch normalization, residual connections, and dropout. Figure 2.10 [2] illustrates all elements in a TCN.



(a)



(b)



(c)

Figure 2.10 The element architectures in a TCN.

In Figure 2.10 (a), the dilation factors $d = 1, 2{,}4$ with filter size $k = 3$ are used to operate dilated causal convolution. The receptive field can capture all the input sequences. Figure 2.10 (b) is a TCN residual block, and Figure 2.10 (c) is an example of residual connection in a TCN.

2.2.4 Ensemble Learning Neural Networks

Ensemble learning is a method to improve performance by incorporating predictions from multiple different models on the same dataset. The method is popular to reduce high variance in neural networks. Combining multiple models adds bias that counters the variance of a single trained model. It makes the predictions less sensitive to a particular part of training data and can lead to better predictions. We will review ensemble-based models used in this dissertation.

2.2.4.1 Random Multimodel Deep Learning (RMDL)

K. Kowsari et al. [3] introduced a novel deep learning technique for classification called Random Multimodel Deep Learning (RMDL). The model can be used for any classification task. Figure 2.11 [25] illustrates the proposed architecture using deep RNN, deep CNN, and deep feedforward neural network (DNN). The name *"random"* indicates the randomness of generating the number of layers and nodes for each model (e.g., nine random models are constructed from 3 RNNs, 3 CNNs, and 3 DNNs, where each model is unique because of random creation).

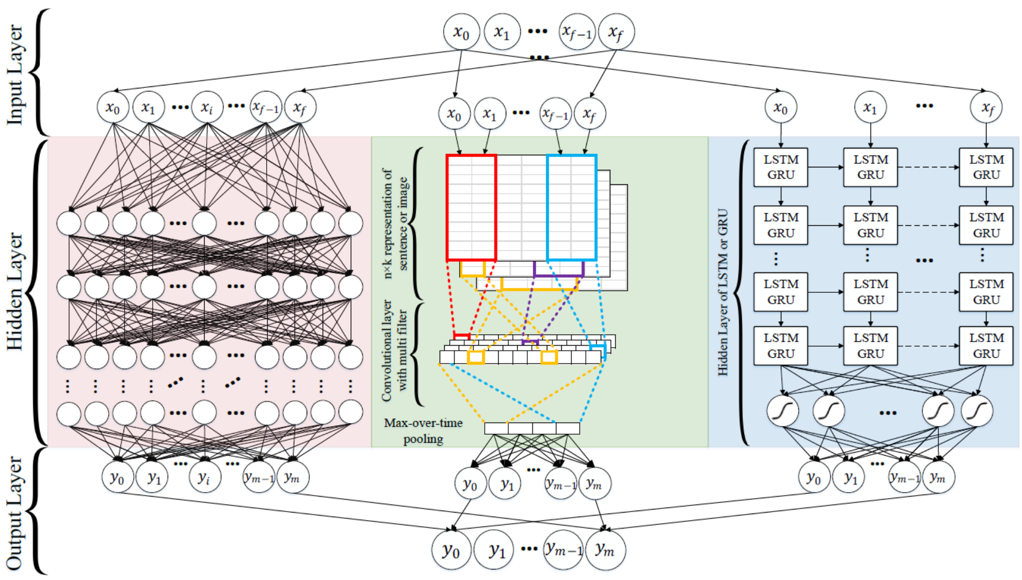


Figure 2.11 The RMDL architecture for classification.

The final prediction is obtained by majority voting on the output of each model. The idea is if k models do not fit well on specific data set, then the RMDL with n models can ignore them if and only if $n > k$.

2.2.4.2 Ensemble Deep Random Vector Functional Link (edRVFL)

In 2019, Rakesh Katuwal et al. [4] proposed an ensemble deep Random Vector Functional Link (edRVFL) that is developed based on RVFL network [12], deep neural network, and ensemble learning. In edRVFL, the deep version of RVFL (dRVFL) and ensemble learning method are constructed. The model uses intermediate features for making the final prediction. Then, the ensemble is acquired by training a single edRVFL once. The training cost should be slightly higher than that of a single dRVFL but lower than training several independent dRVFL models [4]. As illustrated in Figure 2.12 [4], each hidden layer is calculated as follows:

$$\text{The first hidden layer:} \quad H^{(1)} = g(XW^{(1)}) \tag{24}$$

$$\text{For every hidden layer } L > 1\text{:} \quad H^{(L)} = g([H^{(L-1)} X] W^{(L)}) \tag{25}$$

where the output weights β_{ed} are solved independently using Equation (24) and (25) to produce output for each model. At last, the final prediction is obtained by majority voting or averaging of those output models.

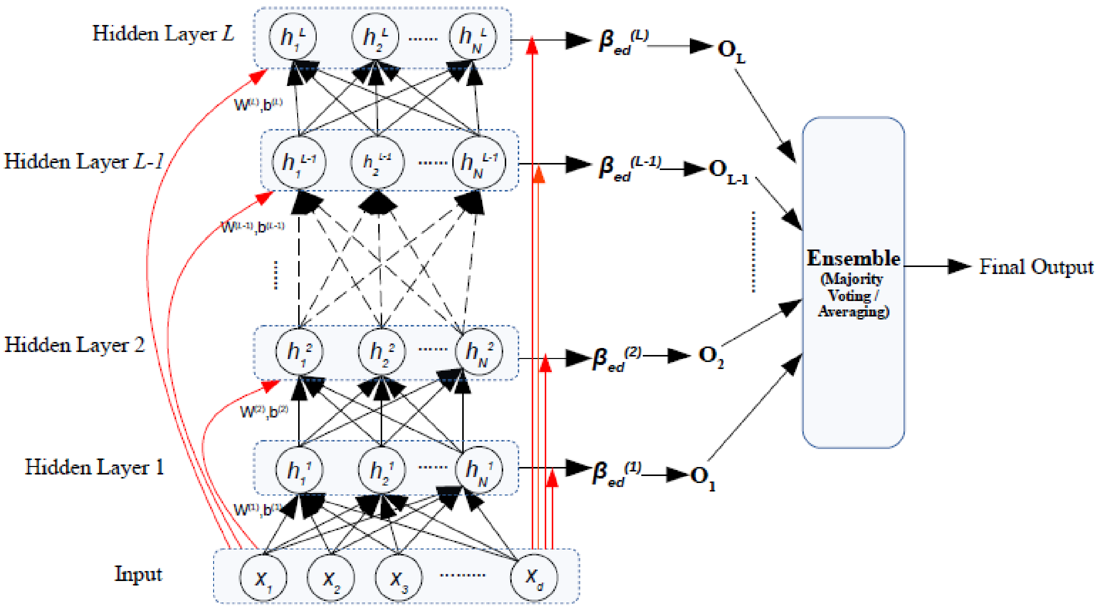


Figure 2.12 The architecture of the edRVFL network.

In edRVFL, activations are used as the important hyperparameters to be tuned. The five common activation functions are sigmoid (Equation (4)), ReLU (Equation (5)), radial basis function or radbas (Equation (26)), sine (Equation (27)), and Scaled Exponential Linear Unit or SELU (Equation (28)), as follows:

$$f(x) = \exp(-x^2) \tag{26}$$

$$f(x) = \sin(x) \tag{27}$$

$$f(x) = \begin{cases} \alpha x, & x \geq 0 \\ \alpha\beta(\exp(x) - 1), & x < 0 \end{cases} \tag{28}$$

where α and β are the predefined constants (α = 1.67326324 and β = 1.05070098).

Chapter 3

The Proposed Deep Learning Models

In this chapter, we present the deep learning models used for the experiment. The chapter covers seven models based on four basic architectures used for the text classification task.

3.1 Feedforward-based Model

3.1.1 Shallow Neural Network (SNN)

The word "shallow" represents a feedforward neural network with no more than four hidden layers. We present three simple models with different numbers of units defined. The first model, SNN-a, has 50 neurons and one hidden layer (Figure 3.1 (a)). Then, we add 50 more neurons in the second model, SNN-b, with still one hidden layer (Figure 3.1 (b)). Finally, we add another hidden layer and neurons in the third model SNN-c (Figure 3.1 (c)). All models use dropout to prevent overfitting. The output layer can be one neuron for binary classification and multiple neurons for multi-class classification. The input features can be either from the bag-of-words or word embedding method. The implementation code can be inspected in Appendix A.

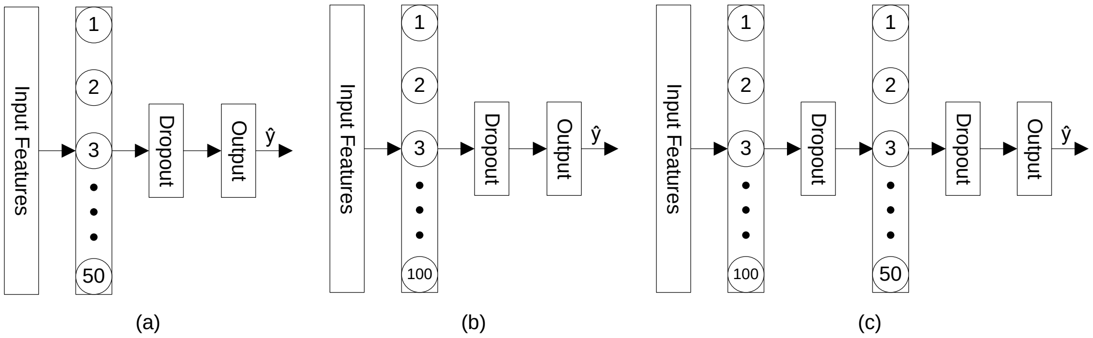


Figure 3.1 The proposed three models of shallow neural network (SNN-a/b/c).

Table 3.1 The hyperparameters of the proposed SNN-a/b/c models.

Model	Hidden layers	Number of neurons	Dropout	Activation function
SNN-a	1	50	0.5	ReLU
SNN-b	1	100		
SNN-c	2	(100; 50)		

3.2 CNN-based Model

3.2.1 1D CNN (Baseline Model)

The 1D CNN is our baseline model representing a neural network with a one-dimensional convolution layer, 100 filters, and a one-dimensional pooling layer. The model is heavily inspired by the work of Yoon Kim [1]. However, instead of going straight to the output after being flattened, we add one fully connected layer with ten neurons and dropout. Since convolution operation works as a feature extractor, the only input feature will only be based on word embeddings. The hyperparameters to be tuned are the kernel size and activation function. The model is illustrated in Figure 3.2. The implementation code can be inspected in Appendix B.

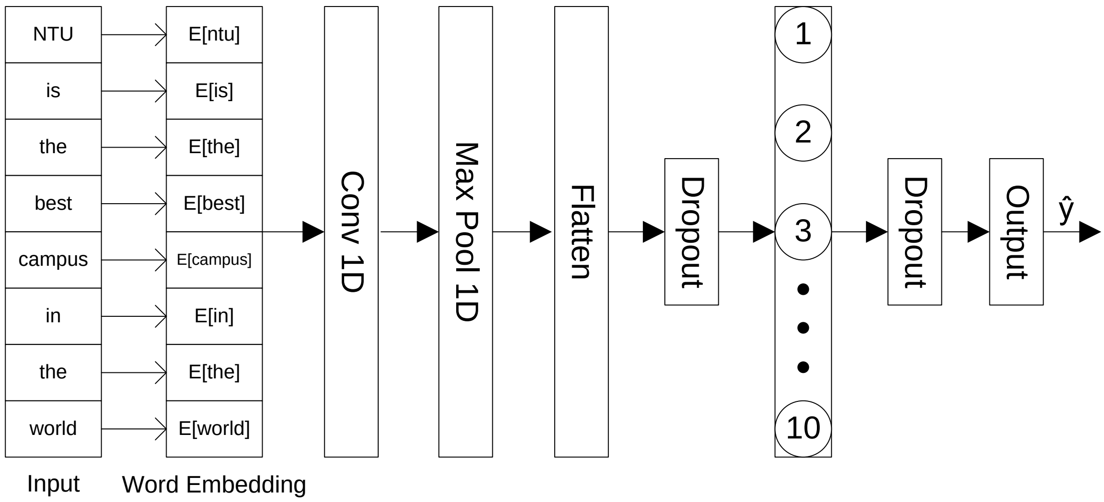


Figure 3.2 The proposed CNN model.

Table 3.2 The hyperparameters of the proposed CNN model.

Filters	Kernel size	Activation function	Dropout	Constraints
100	1 - 6	ReLU, Tanh	0.5	MaxNorm of 3

3.2.2 TCN

The proposed TCN model is inspired by Christof Henkel [14], one of the grandmasters on Kaggle. The model consists of two TCN blocks stacked with the kernel size of 3 and dilation factors of 1, 2, and 4. Each block's result will take the form of a sequence. The final sequence is then passed to two different global pooling layers. Next, both results are concatenated and passed into a dense layer of 16 neurons and pass to the output. The first TCN block contains 128 filters, and the second block uses 64 filters. The input features will be based on Word Embedding.

The TCN model used for the experiment is illustrated in Figure 3.3. The implementation code can be inspected in Appendix C.

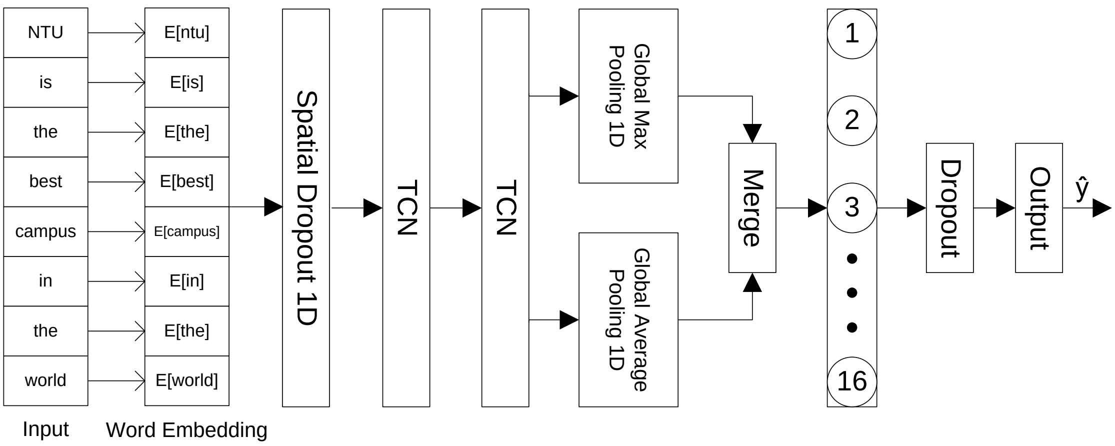


Figure 3.3 The proposed TCN model.

Table 3.3 The hyperparameters of the proposed TCN model.

TCN block	Filters	Kernel size	Dilation factors	Activation function	Dropout
1st	128	3	[1, 2, 4]	ReLU, tanh	0.1
2nd	64				

3.3 RNN-based Model

3.3.1 BiGRU/BiLSTM

To consider information from both earlier and later in a sequence, we design a bidirectional RNN-based model. The model suits text classification because normally we always obtain a whole document as input to make a prediction. Figure 3.4 illustrates the architecture of bidirectional RNN with word embedding as input feature extraction.

Notice that instead of processing a whole sentence like in CNN, the BiRNN model will process word by word in both forward and backward directions. The RNN-based block will have 64 units (neurons). We will implement the model using both GRU and LSTM as two different models to compare the performances. We also use a dropout layer again to prevent overfitting. The implementation code can be inspected in Appendix D.

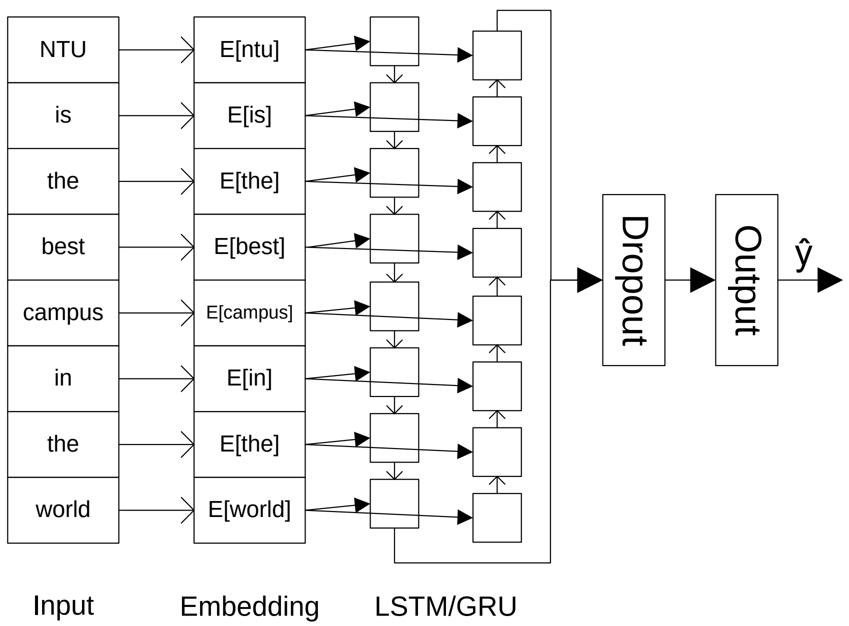


Figure 3.4 The proposed BiGRU/BiLSTM model.

3.3.2 Stacked BiGRU/BiLSTM

The stacked BiRNN-based model is the extended version of the standard BiRNN model proposed in the previous section. Now, instead of one bidirectional layer, we stack another identical layer. The result adds complexity to the previous model in the hope of a better performance result. Notice that the first bidirectional layer returns a sequence. Then, the second one will not return a sequence. It only passes the final information to the output layer. The implementation code can be inspected in Appendix E.

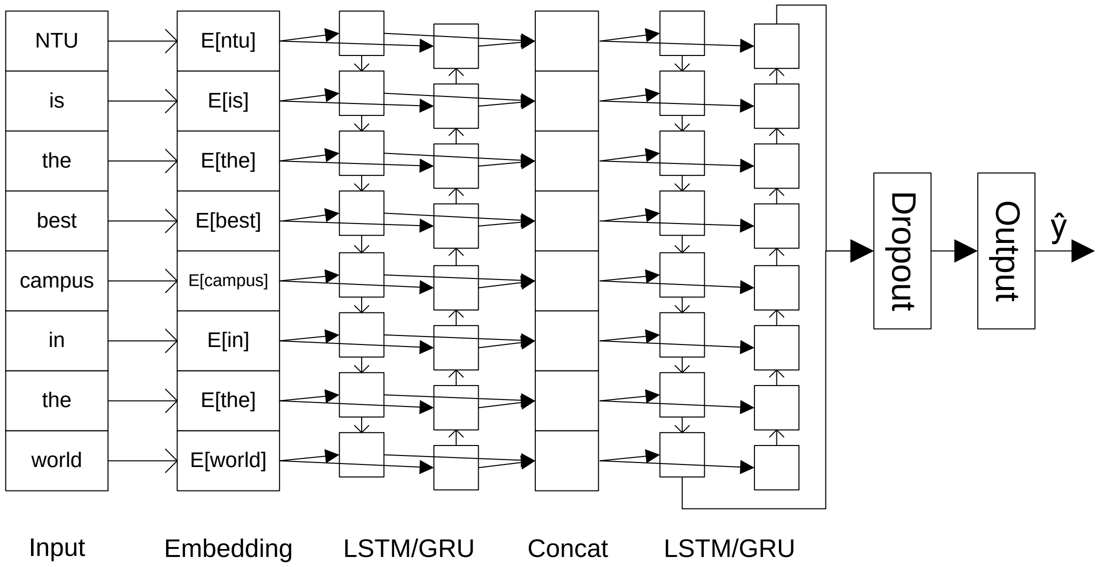


Figure 3.5 The proposed stacked BiGRU/BiLSTM.

Table 3.4 The hyperparameters of the proposed BiGRU/BiLSTM model.

Model	Activation function	Number of units	Dropout
BiGRU/BiLSTM	tanh	64	0.5
Stacked BiGRU/BiLSTM			

3.4 Ensemble Learning-based Model

3.4.1 edRVFL

We will implement the edRVFL model exactly like in Figure 2.12. The only hyperparameters we keep fixed is the number of hidden layers, which is 10. The other hyperparameters, such as regularization parameters, activation function, and the number of neurons will be tuned. The model can take input features based on either bag-of-words or word embedding. The implementation code can be inspected in Appendix G.

Table 3.5 The hyperparameters of the proposed edRVFL model.

N-layers	N-neurons	Regularization	Activation function
10	3:20:203	2^(-5:1:14)	ReLU, sigmoid, SELU, radbas, sine

3.4.2 Ensemble CNN-GRU

We implement an ensemble learning-based model by combining 1D CNN in section 3.2.1 with a single BiGRU in section 3.3.1. The 1D CNN has been proven to work well on text classification despite only a little parameter tuning. On the other hand, BiGRU works well on temporal data by taking both earlier and later information in the sequence. We will see how this combination affects the model accuracy in the experiment. Therefore, the hyperparameters of this model will be the same as in Table 3.2 and Table 3.4. The implementation code can be inspected in Appendix F.

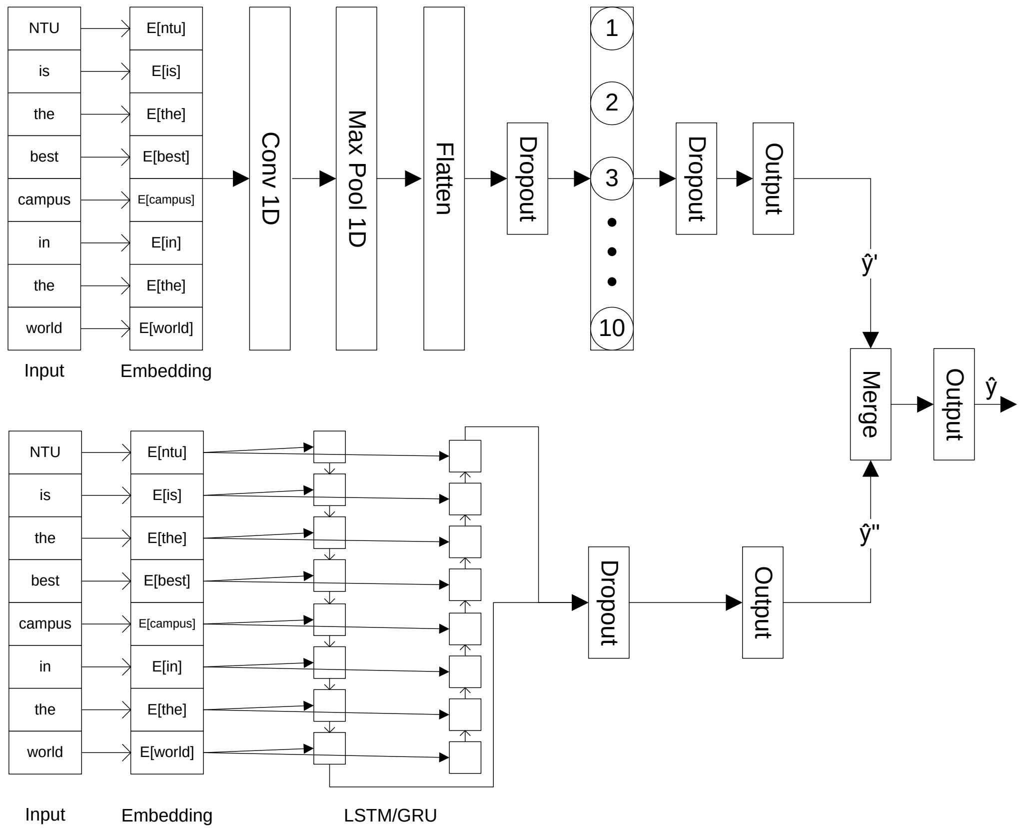


Figure 3.6 The proposed ensemble learning model with CNN and BiGRU combined.

3.5 Experiment Summary

In short, we will train various deep learning models based on seven main architectures using two different feature extractions. We will evaluate each model on five different text classification datasets. In the end, we will also compare each performance with the SOTA models as the benchmarks. Figure 3.7 summarizes the series of experiments we will perform in this dissertation.

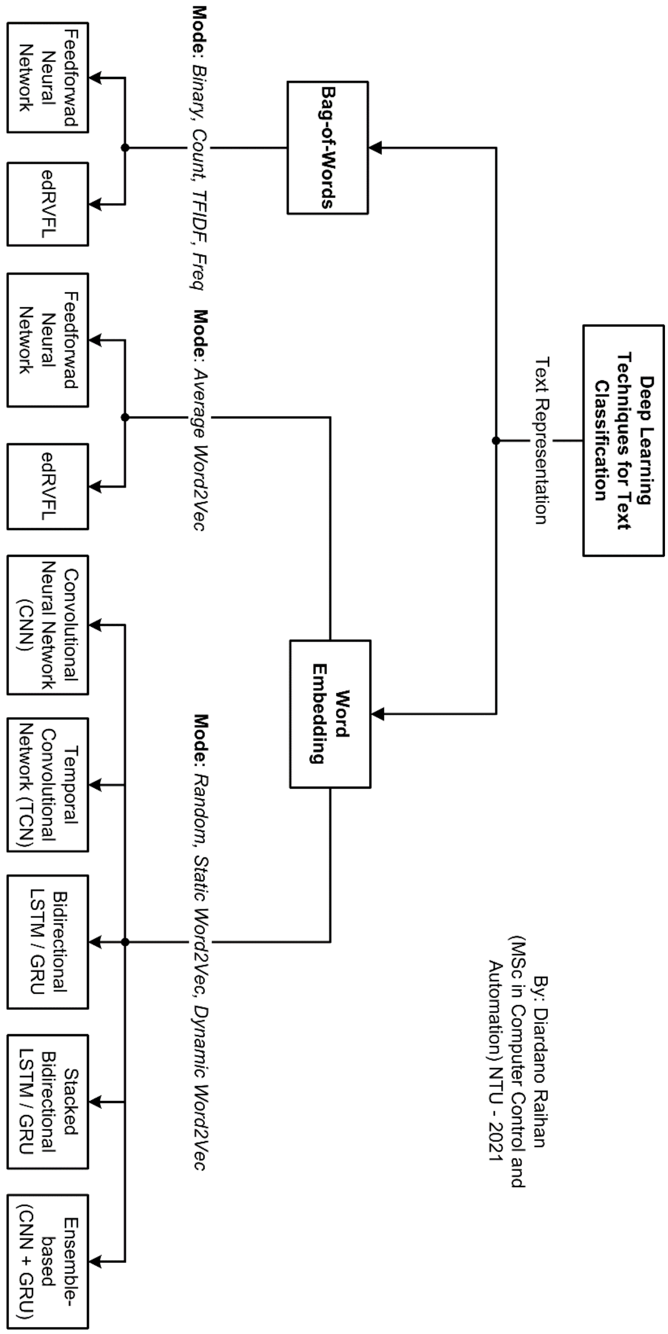


Figure 3.7 The proposed deep learning models to experiment.

Chapter 4

Experiment

This chapter will present the datasets used for experiments. The chapter also discusses the text pre-processing methods for representing text data. In the end, we summarize the BoW and word embedding variants used for each model to experiment.

4.1 Datasets

We use five text classification datasets with the summary statistics are in Table 4.1.

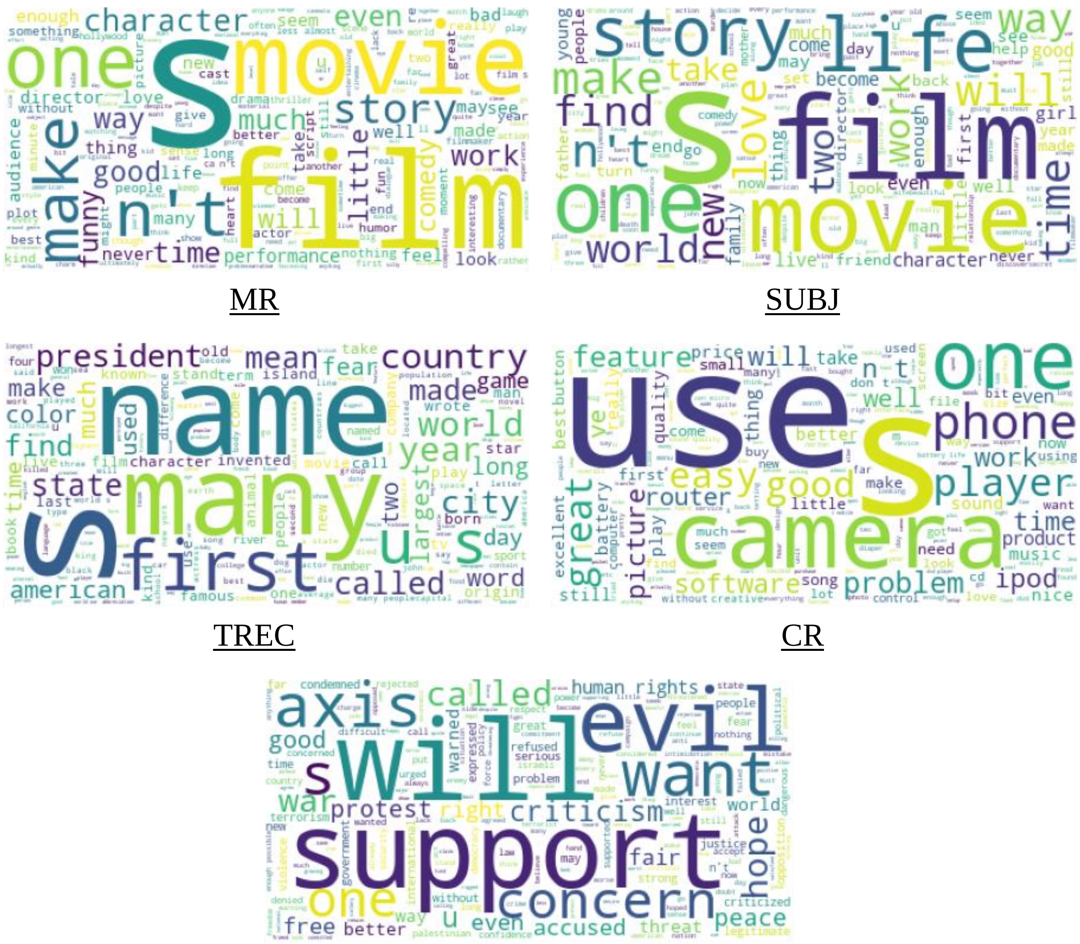
Table 4.1 Dataset statistics after tokenization.

Dataset	Classes	Average sentence length	Dataset size	Vocab size	Number of words present in Word2Vec	Test Size
MR	2	20	10662	18758	16448	CV
SUBJ	2	23	10000	21322	17913	CV
TREC	6	10	5952	8759	9125	500
CR	2	19	3775	5334	5046	CV
MPQA	2	3	10606	6234	6083	CV

- **MR**: Movie Reviews – classifying a review as positive or negative [17].
- **SUBJ**: Subjectivity – classifying a sentence as subjective or objective [18].
- **TREC**: Text REtrieval Conference – classifying a question into six categories (a person, location, numeric information, etc.) [19].
- **CR**: Customer Reviews – classifying a product review (cameras, MP3s, etc.) as positive or negative [20].
- **MPQA**: Multi-Perspective Question Answering – opinion polarity detection [21].

Notice in Table 4.1, the test size CV stands for cross-validation. It indicates the original dataset does not have a standard train/test split. Hence, we use a 10-fold CV. The AcademiaSinicaNLPLab [22] repository provides access to all these datasets.

Figure 4.1 shows the word clouds for each text dataset. The bigger the word, the more frequent that word appears inside the data.



MR

SUBJ

TREC

CR

MPQA

Figure 4.1 The word clouds of the text datasets.

4.2 BoW Scoring

We will use the bag-of-words method to extract information from the text by creating vocabulary and scoring each word presented in the input data. The first step to creating a good vocabulary is by cleaning the text. There is no universal answer for this cleaning method. The procedure we follow for representing text using BoW is as follows:

- Split the sentence into words (tokens) by whitespace;
- Remove punctuation from each word;
- Filter out the stop words;
- Filter out the short token;
- Stem the token;
- Create a list of vocabulary;
- Convert the input data into BoW representation based on words listed in the defined vocabulary.
- Compare the model performance for each word scoring option.

Notice that for each model trained using BoW representation, the model will produce four different results due to four-word scoring options: binary, count, freq, and TF-IDF.

4.3 Word Embedding (WE)

In this dissertation, we experiment with four variants of word embedding.

- **Model-rand**. The model uses an embedding layer where the word vectors are randomly initialized and corrected during training [1].
- **Model-static**. The model uses pre-trained word embedding called Word2Vec vectors trained on Google news containing 100 billion words. The Word2Vec has 300-dimensional vectors and uses CBOW architecture [7]. The vectors are kept static during training. The vectors for unknown words are randomly initialized using a generic normal distribution.
- **Model-dynamic**. Same as above, but the vectors are modified during training, not static.
- **Model-avg**. The model uses the average of vectors from the pre-trained word embedding to get the input context. Hence, the size of input features will be the same as the size of the vector dimension used in the Word2Vec, 300.

Fortunately, the text cleaning for word embedding is simple compared to the cleaning process of BoW. Generally, we do not need to do the stemming and removing the stop words. Besides, although removing punctuation can help, but that is not necessary.

4.4 Training

For all models, the training process is done using an early stopping where the model will stop training before it overfits the training data. Hence, early stopping can minimize overfitting and improve the model generalization. We will treat the epoch numbers as a hyperparameter. The number of epochs for each model varies, but the *"patience"* parameter set for early stopping is typically between 5 – 10 epochs. The training is also performed using Adam optimizer over the shuffled mini-batch size of 32 or 50.

The model accuracy is calculated by the formula given below:

$$Accuracy = \frac{(True\ Positive + True\ Negative) \times 100\%}{(True\ Positive + False\ Positive + True\ Negative + False\ Negative)} \tag{29}$$

and the variants of feature extractions used by all the models are shown in Table 4.2.

Table 4.2 The variants of feature extractions used by all the proposed models.

Model	Bag-of-Words	Word Embedding			
		-rand	-static	-dynamic	-avg
SNN-a/b/c	✓	-	-	-	✓
edRVFL	✓	-	-	-	✓
1D CNN	-	✓	✓	✓	-
TCN	-	✓	✓	✓	-
BiGRU/BiLSTM	-	✓	✓	✓	-
Stacked BiGRU/BiLSTM	-	✓	✓	✓	-
Ensemble CNN-GRU	-	✓	✓	✓	-

Chapter 5

Evaluation

This chapter showcases the results obtained from the proposed models to do text classification tasks on five datasets. We present all the results followed by the discussion in two separate sections.

5.1 Result

Before we compare all the models against benchmarks, we will show the model performances that used bag-of-words and average Word2Vec. We will use accuracy and rank as comparison metrics. The rank will be calculated based on the accuracy on each dataset. In the case there are ties, we average their ranks.

5.1.1 Bag-of-Words

This section shows the model results using BoW for representing text on the classification task. The models used are the shallow neural networks and edRVFL.

5.1.1.1 Shallow Neural Networks

Table 5.1 The SNN model performance using BoW.

SNN Model	**Word Scoring**	**Accuracy (%)**				
		MR	*SUBJ*	*TREC*	*CR*	*MPQA*
SNN-a	Binary	77.3	90.2	75	79.4	**86.0**
	Count	77.0	90.1	74.6	79.6	85.6
	TF-IDF	77.0	90.6	73.6	**79.7**	85.6
	Freq	77.3	**90.8**	74	79.4	85.4
SNN-b	Binary	77.2	90.1	75.6	78.9	85.5
	Count	77.1	90.1	75	79.5	85.5
	TF-IDF	**77.4**	90.6	72.8	79.6	85.3
	Freq	77.0	90.5	74.2	79.1	85.5
SNN-c	Binary	77.0	90.0	74.4	79.4	85.6

	Count	76.6	90.0	**76.2**	79.5	85.4
	TF-IDF	76.8	90.1	72.8	79.6	85.2
	Freq	77.2	90.3	72.6	**79.7**	85.4

Table 5.2 The average rank values for each BoW word scoring in SNN models.

BoW Word Scoring	Rank Average per Dataset (based on the combined SNN-a/b/c accuracies)					Average Rank Values
	MR	*SUBJ*	*TREC*	*CR*	*MPQA*	
Binary	1.5	3	2	4	1	**2.3**
Count	4	4	1	2	2	2.6
TF-IDF	3	2	4	1	4	2.8
Freq	1.5	1	3	3	3	**2.3**

5.1.1.2 edRVFL

Table 5.3 The edRVFL model performance using BoW.

Dataset: MR						
Activation	**Accuracy (%) per Word Scoring**				**Average Accuracy (%)**	**Rank Average**
	Binary	*Count*	*TFIDF*	*Freq*		
ReLU	75.4	76.1	72.3	75.9	74.9	5
Sigmoid	76.0	**76.2**	72.4	**76.2**	75.2	1
SELU	75.5	75.9	72.3	76.1	75.0	3.5
Radbas	75.6	75.9	72.4	76.0	75.0	3.5
Sine	75.8	**76.2**	72.4	75.8	75.1	2
Dataset: SUBJ						
Activation	**Accuracy (%) per Word Scoring**				**Average Accuracy (%)**	**Rank Average**
	Binary	*Count*	*TFIDF*	*Freq*		
ReLU	88.7	88.8	84.2	**89.4**	87.8	2.5
Sigmoid	88.9	88.8	84.2	89.3	87.8	2.5
SELU	88.8	88.9	84.1	**89.4**	87.8	2.5
Radbas	88.7	88.8	84.4	**89.4**	87.8	2.5

Sine	88.6	88.7	84.3	89.3	87.7	5
Dataset: TREC						
Activation	**Accuracy (%) per Word Scoring**				**Average Accuracy (%)**	**Rank Average**
	Binary	*Count*	*TFIDF*	*Freq*		
ReLU	74.0	74.8	71.6	60.0	70.1	4
Sigmoid	74.8	75.0	71.0	60.0	70.2	3
SELU	**75.2**	75.0	71.2	73.2	73.7	1
Radbas	74.0	74.4	69.4	60.2	69.5	5
Sine	75.0	**75.2**	71.0	59.8	70.3	2
Dataset: CR						
Activation	**Accuracy (%) per Word Scoring**				**Average Accuracy (%)**	**Rank Average**
	Binary	*Count*	*TFIDF*	*Freq*		
ReLU	**78.0**	77.3	75.3	77.5	77.0	2.5
Sigmoid	78.0	77.5	75.0	77.4	77.0	2.5
SELU	77.4	77.2	74.9	77.4	76.7	5
Radbas	**78.0**	77.6	75.9	77.6	77.3	1
Sine	76.7	77.6	75.5	77.4	76.8	4
Dataset: MPQA						
Activation	**Accuracy (%) per Word Scoring**				**Average Accuracy (%)**	**Rank Average**
	Binary	*Count*	*TFIDF*	*Freq*		
ReLU	**85.0**	84.9	84.4	84.6	84.7	2.5
Sigmoid	84.7	84.9	84.4	84.6	84.7	2.5
SELU	84.6	84.9	84.2	84.7	84.6	5
Radbas	84.6	84.7	84.6	84.7	84.7	2.5
Sine	84.7	84.9	84.3	84.8	84.7	2.5

Using Table 5.3 as the guidance, we can compile the rank average on each dataset to create average rank values as follows:

Table 5.4 The average rank values for each activation effect in the edRVFL-BoW model.

Activation	**Rank Average per Dataset**					**Average Rank Values**
	MR	*SUBJ*	*TREC*	*CR*	*MPQA*	
ReLU	5	2.5	4	2.5	2.5	3.3
Sigmoid	1	2.5	3	2.5	2.5	**2.3**
SELU	3.5	2.5	1	5	5	3.4
Radbas	3.5	2.5	5	1	2.5	2.9
Sine	2	5	2	4	2.5	3.1

Table 5.5 The average rank values for each BoW word scoring in the edRVFL model.

BoW Word Scoring	**Rank Average per Dataset** (based on the Best Activation Accuracy)					**Average Rank Values**
	MR	*SUBJ*	*TREC*	*CR*	*MPQA*	
Binary	3	2.5	1.5	1	1	**1.8**
Count	1.5	2.5	1.5	2.5	2	2
TF-IDF	4	4	4	4	4	4
Freq	1.5	1	3	2.5	3	2.2

5.1.2 Average Word2Vec

This section shows the model results using the average word embedding Word2Vec for representing text on the classification task. The models used are the shallow neural networks and edRVFL.

5.1.2.1 Shallow Neural Networks

Table 5.6 The SNN model performance using the average of Word2Vec.

SNN Model	**Accuracy (%)**				
	MR	*SUBJ*	*TREC*	*CR*	*MPQA*
SNN-a	78.1	91.4	85.0	80.1	87.5
SNN-b	78.0	91.5	85.6	80.5	87.5
SNN-c	**78.3**	**91.6**	**85.8**	**80.5**	**87.6**

Table 5.7 The average rank values for each SNN model using the average of Word2Vec.

SNN Model	**Rank Average**					**Average Rank Values**
	MR	*SUBJ*	*TREC*	*CR*	*MPQA*	
SNN-a	2	3	3	3	2.5	2.7
SNN-b	3	2	2	1.5	2.5	2.2
SNN-c	1	1	1	1.5	1	**1.1**

5.1.2.2. edRVFL

Table 5.8 The edRVFL model performance using the average Word2Vec vectors.

Activation	**Accuracy (%)**				
	MR	*SUBJ*	*TREC*	*CR*	*MPQA*
ReLU	76.4	**90.6**	81.6	78.2	86.6
Sigmoid	76.7	90.5	**83.6**	**78.5**	86.6
SELU	**77.0**	90.4	82.0	78.2	**86.7**
Radbas	76.4	90.2	81.6	78.2	86.5
Sine	76.6	90.2	81.8	78.2	86.5

Table 5.9 The average rank values for each activation in the edRVFL-avg model.

Activation	**Rank Average**					**Average Rank Values**
	MR	*SUBJ*	*TREC*	*CR*	*MPQA*	
ReLU	4.5	1	4.5	3.5	2.5	3.2
Sigmoid	2	2	1	1	2.5	**1.7**
SELU	1	3	2	3.5	1	2.1
Radbas	4.5	4.5	4.5	3.5	4.5	4.3
Sine	3	4.5	3	3.5	4.5	3.7

We have highlighted several best models with the highest accuracy from edRVFL and shallow neural networks. Now, we can compare their performances with the proposed models that used word embeddings and the SOTA benchmark models.

5.1.3 Word Embedding

Table 5.10 shows the final comparison for each model performance. We also include the SOTA benchmark models (in the green shading) for further observation. Note that we only include the best results for the models that use the bag-of-words and average word embedding (SNN and edRVFL), as shown in Table 5.1, 5.3, 5.5, and 5.8.

In Table 5.10, the models with purple color managed to beat the baseline, while the red ones do not. The rest of the models with the black color do not entirely surpass the baseline, indicating only a few datasets achieve better performances than the baseline. From here, we can calculate the average accuracy margin of all models to the baseline.

Table 5.10 The proposed models against benchmarks (in the green shading).

Model	Accuracy (%)				
	MR	*SUBJ*	*TREC*	*CR*	*MPQA*
edRVFL-BoW	76.2	89.4	75.2	78.0	85.0
edRVFL-avg	77.0	90.6	83.6	78.5	86.7
SNN-a/b/c-BoW	77.4	90.8	76.2	79.7	86.0
SNN-c-avg	78.3	91.6	85.8	80.5	87.6
1D CNN-rand ***(baseline)***	*77.6*	*92.05*	*89.8*	*80.4*	*86.4*
1D CNN-static	79.0	92.51	92.2	81.4	**88.6**
1D CNN-dynamic	79.4	92.8	91.6	82.2	87.5
TCN-rand	77.3	91.4	90.0	81.2	86.3
TCN-static	**80.3**	92.3	**93.6**	**83.9**	88.3
TCN-dynamic	80.0	92.4	91.8	82.9	88.1
BiLSTM-rand	77.6	91.9	88.4	80.6	86.3
BiLSTM-static	79.5	92.5	90.4	81.7	88.2
BiLSTM-dynamic	79.8	92.6	88.8	81.8	88.0
BiGRU-rand	77.2	92.2	89.0	80.1	86.1
BiGRU-static	79.5	92.3	91.8	82.4	88.1
BiGRU-dynamic	79.2	**93.0**	90.6	81.6	88.1
Stacked BiLSTM-rand	77.7	91.9	89.6	79.7	86.1
Stacked BiLSTM-static	79.4	92.2	91.6	80.9	88.1
Stacked BiLSTM-dynamic	80.0	92.5	88.4	81.7	88.1
Stacked BiGRU-rand	76.9	92.3	89.2	80.1	85.9
Stacked BiGRU-static	79.6	92.3	92.0	81.5	88.1
Stacked BiGRU-dynamic	79.5	92.7	91.0	81.6	88.0
Ensemble CNN-GRU-rand	77.0	91.7	88.0	80.9	86.3
Ensemble CNN-GRU-static	79.8	92.7	93.0	82.5	88.4
Ensemble CNN-GRU-dynamic	79.4	92.6	89.6	82.4	88.0
CNN-multichannel (Yoon Kim, 2014) [1]	81.1	93.2	92.2	85.0	89.4
SuBiLSTM (Siddhartha Brahma, 2018) [15]	**81.4**	93.2	89.8	86.4	**90.7**
SuBiLSTM-Tied (Siddhartha Brahma, 2018) [15]	81.6	93.0	90.4	86.5	90.5
USE_T+CNN (Cer et al., 2018) [16]	81.2	**93.6**	**98.1**	**87.5**	87.3

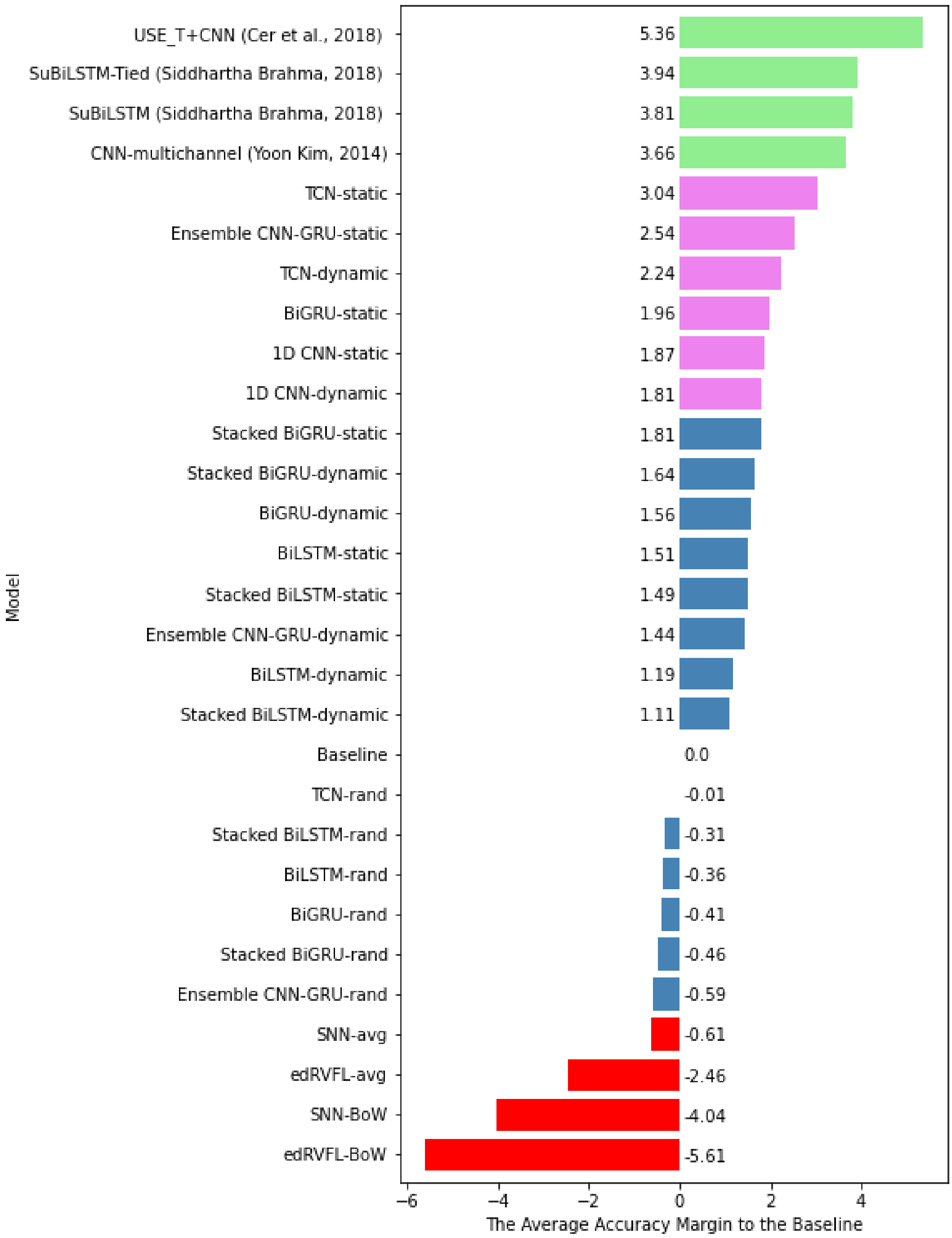


Figure 5.1 The average accuracy margin of the models to the baseline on the 5 datasets.

In Figure 5.1, the green bar represents the benchmark model. The purple bar depicts the top six proposed models that beat the baseline. Finally, the red bar is the proposed model with the lowest accuracy margin. The minus (-) sign indicates the model has much lower accuracy than higher ones in all datasets with the baseline as the reference.

We can calculate the average rank values derived from Table 5.10 as follows:

Table 5.11 The average rank values (ARV) for each model against benchmarks.

Model	**Rank Average**					**ARV**
	MR	*SUBJ*	*TREC*	*CR*	*MPQA*	
edRVFL-BoW	29.0	29.0	29.0	29.0	29.0	29.0
edRVFL-avg	26.5	28.0	27.0	28.0	20.0	25.9
SNN-a/b/c-BoW	23.0	27.0	28.0	26.5	27.0	26.3
SNN-c-avg	19.0	25.0	26.0	22.0	17.0	21.8
1D CNN-rand ***(baseline)***	21.5	21.0	16.5	23.0	21.0	20.6
1D CNN-static	18.0	11.0	4.5	17.0	4.0	**10.9**
1D CNN-dynamic	15.0	6.0	9.5	10.0	18.0	11.7
TCN-rand	24.0	26.0	15.0	18.0	23.0	21.2
TCN-static	5.0	16.5	2.0	5.0	6.0	**6.9**
TCN-dynamic	6.5	14.0	7.5	6.0	10.5	**8.9**
BiLSTM-rand	21.5	22.5	23.5	21.0	23.0	22.3
BiLSTM-static	12.0	12.5	13.5	12.5	7.0	**11.5**
BiLSTM-dynamic	8.5	9.5	22.0	11.0	15.0	13.2
BiGRU-rand	25.0	19.5	21.0	24.5	25.5	23.1
BiGRU-static	12.0	16.5	7.5	8.5	10.5	**11.0**
BiGRU-dynamic	17.0	4.5	12.0	14.5	10.5	11.7
Stacked BiLSTM-rand	20.0	22.5	18.5	26.5	25.5	22.6
Stacked BiLSTM-static	15.0	19.5	9.5	19.5	10.5	14.8
Stacked BiLSTM-dynamic	6.5	12.5	23.5	12.5	10.5	13.1
Stacked BiGRU-rand	28.0	16.5	20.0	24.5	28.0	23.4
Stacked BiGRU-static	10.0	16.5	6.0	16.0	10.5	11.8
Stacked BiGRU-dynamic	12.0	7.5	11.0	14.5	15.0	12.0
Ensemble CNN-GRU-rand	26.5	24.0	25.0	19.5	23.0	23.6
Ensemble CNN-GRU-static	8.5	7.5	3.0	7.0	5.0	**6.2**
Ensemble CNN-GRU-dynamic	15.0	9.5	18.5	8.5	15.0	13.3
CNN-multichannel (Yoon Kim, 2014) [1]	4.0	2.5	4.5	4.0	3.0	**3.6**
SuBiLSTM (Siddhartha Brahma, 2018) [15]	2.0	2.5	16.5	3.0	1.0	**5.0**
SuBiLSTM-Tied (Siddhartha Brahma, 2018) [15]	1.0	4.5	13.5	2.0	2.0	**4.6**
USE_T+CNN (Cer et al., 2018) [16]	3.0	1.0	1.0	1.0	19.0	**5.0**

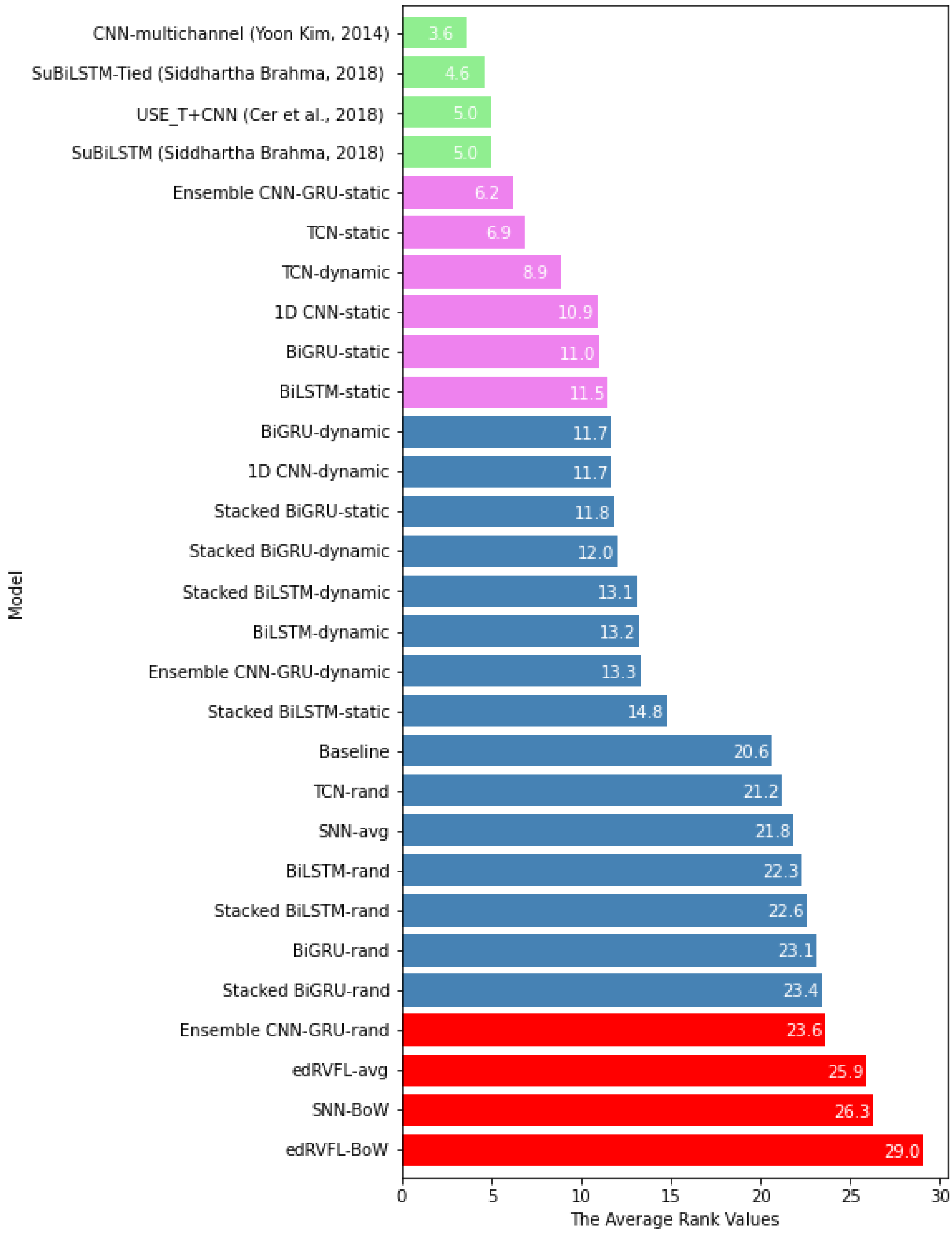


Figure 5.2 The average rank values for each model against benchmarks

Using the same scenario as in Figure 5.1, Figure 5.2 shows the top six models (the violet bar) with high average ranks and can compete with the benchmarks (the green bar). We will discuss all of these results in the next section.

5.2 Discussion

It is worth mentioning again that CNN can work well as a baseline despite having a little parameter tuning. From more than 25 model experiments, only eleven models completely surpass the baseline and achieve better accuracies. It shows that a 1D CNN with the word vector randomly initialized is a robust baseline for text classification.

Furthermore, we have some interesting observations in the experiments. For example, we were surprised by the effect of pre-trained word embedding on the model performance. Almost all models increase their accuracies with a high margin. In this section, we will discuss any possible insights obtained from the results.

5.2.1 Shallow vs. Deep Neural Network

Table 5.1 shows the SNN performance with BoW. We can see that the accuracy tends to be lower when we add more neurons or hidden layers to the network (a deeper neural network). The decrease in performance is because the model will try to learn input features that contain many zeros in a large vector representation (sparsity). These zeros result from the BoW method representing any word that does not appear in the input sentence. Hence, adding more layers to the model will make the training process even harder and overfitted.

However, Table 5.6 and 5.7 show that the deeper the model, the better the performance is when word embedding Word2Vec is involved. This is because word embedding keeps the input (sentence) context and converts it into a better dense representation for the model to learn. As a result, adding more neurons or hidden layers will help the model learn more complex functions and uncover more useful information from the input resulting in a better prediction.

5.2.2 EDRVFL Activations

Using Table 5.4 and 5.9, we can create a new table to calculate the final average rank values for the edRVFL activation functions used in this work. The result is as shown in Table 5.12

Table 5.12 The average rank values for activations used in the edRVFL model.

Activation	**Average Rank Values**		**Average Rank Values** (Final)
	BoW	*Average Word2Vec*	
ReLU	3.3	3.2	3.25
Sigmoid	**2.3**	**1.7**	**2**
SELU	3.4	2.1	2.75
Radbas	2.9	4.3	3.6
Sine	3.1	3.7	3.4

We can see that the sigmoid activation function frequently delivers better prediction than other functions in both feature extractors. In some datasets, SELU also performs better compared to the others. The radbas does well in the BoW but performs poorly in the average Word2Vec. The most frequently used activation, ReLU, acts moderately in this work. To sum up, the following is the rank for activations used by edRVFL:

$$Sigmoid \; > \; SELU > ReLU > Radbas > Sine$$

5.2.3 BoW Word Scoring

Using Table 5.2 and 5.5, we can create a new table to calculate the final average rank values for BoW word scoring options. The result is as shown in Table 5.13.

Table 5.13 The average rank values for the BoW methods used in the experiment.

BoW Word Scoring	**Average Rank Values**		**Average Rank Values** (Final)
	SNN	*edRVFL*	
Binary	**2.3**	**1.8**	**2.05**
Count	2.6	2	2.3
TF-IDF	2.8	4	3.4
Freq	2.3	2.2	2.25

The binary method shows as the best word scoring method in this work. Both models, SNN and edRVFL, perform better in making a final prediction. This result is quite surprising because the binary is the simplest existing method yet delivers remarkably well than other complex ones, such as TF-IDF.

5.2.4 BoW vs. Word Embedding

The models with BoW in this experiment cannot do much despite having so many hyperparameter tuning. For example, the simple architecture and computational process offered by edRVFL teases us to do intensive model training with various ranges of hyperparameter values. However, both edRVFL and SNN still cannot beat the baseline. The large numbers of text data will make the vocabulary of BoW extensive. Hence, the input features will be in sparse form, presenting a bit of information over many zeros. This text representation makes the model harder to train to achieve a better result. Unless we specify the vocabulary size not big enough or work with a small corpus, BoW cannot be a reliable option.

On the other hand, both models perform better when using word embedding. By only taking the average of Word2Vec to obtain N-dimensional feature inputs, the model can have a very steep increase in accuracy up to 10%. For example, both edRVFL and SNN suddenly jump from 75.2 and 76.2 to 83.6 and 85.8 in the TREC dataset. These results prove the importance of word embedding as a default feature extractor.

5.2.5 Random vs. Static vs. Dynamic

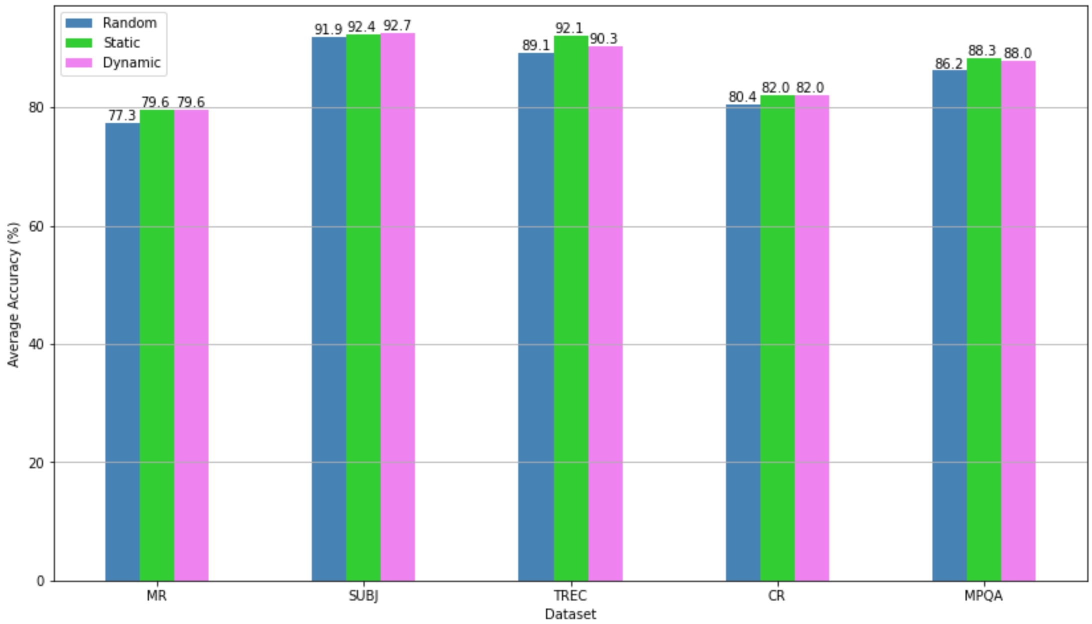


Figure 5.3 The average accuracy between different word embedding modes.

Figure 5.3 illustrates the effect of different word embedding modes on the model performance. As expected, the static word embedding using pre-trained Word2Vec always performs better. The static mode can help any models predict classes more accurately up to 3% average accuracy increase than the random mode.

The dynamic vector representation model will fine-tune the parameters initialized by Word2Vec vectors to learn the meaningful context for each task. Ideally, it will result in better performance than the static one. However, that is not always the case. Although the model can still improve, the change is not significant. In some cases, a model can even have lower accuracy.

In Figure 5.3, the dynamic mode slightly lowers the overall model performance on TREC and MPQA datasets. In Table 5.10, although BiGRU-dynamic offers better performance than its static version in the SUBJ dataset, it decreases performance on the other datasets. This is because the vectors adjust to a specific dataset that can overfit and change the original context derived from Word2Vec.

5.2.6 TCN vs. RNN Model

If we use word embedding, TCN is more effective than RNN-based models like LSTM or GRU, as shown in Table 5.14 and Table 5.15. In four of five datasets, TCN outclasses all the RNN architectures with an excellent accuracy margin. On the other dataset, the TCN accuracy is still high and close to the highest ones. TCN-static and -dynamic sit as the top models, followed by BiLSTM-static, BiGRU-static, and Stacked BiGRU-static.

Simply put, TCN is the best model not only compared to the RNN family but also to the other models in capturing information to make a stable prediction. The only type of model that can challenge TCN in this experiment is the ensemble-based model.

Table 5.14 The performance of TCN vs. RNN-based models.

Model	**Accuracy (%)**				
	MR	*SUBJ*	*TREC*	*CR*	*MPQA*
TCN-rand	77.3	91.4	90.0	81.2	86.3
TCN-static	**80.3**	92.3	**93.6**	**83.9**	**88.3**

TCN-dynamic	80.0	92.4	91.8	82.9	88.1
BiLSTM-rand	77.6	91.9	88.4	80.6	86.3
BiLSTM-static	79.5	92.5	90.4	81.7	88.2
BiLSTM-dynamic	79.8	92.6	88.8	81.8	88.0
BiGRU-rand	77.2	92.2	89.0	80.1	86.1
BiGRU-static	79.5	92.3	91.8	82.4	88.1
BiGRU-dynamic	79.2	**93.0**	90.6	81.6	88.1
Stacked BiLSTM-rand	77.7	91.9	89.6	79.7	86.1
Stacked BiLSTM-static	79.4	92.2	91.6	80.9	88.1
Stacked BiLSTM-dynamic	80.0	92.5	88.4	81.7	88.1
Stacked BiGRU-rand	76.9	92.3	89.2	80.1	85.9
Stacked BiGRU-static	79.6	92.3	92.0	81.5	88.1
Stacked BiGRU-dynamic	79.5	92.7	91.0	81.6	88.0

Table 5.15 The average rank values (ARV) for TCN vs. RNN-based models.

Model	**Rank Average**					**ARV**
	MR	*SUBJ*	*TREC*	*CR*	*MPQA*	
TCN-rand	13	15	9	10	11.5	11.7
TCN-static	1	8.5	1	1	1	**2.5**
TCN-dynamic	2.5	6	3.5	2	5.5	3.9
BiLSTM-rand	12	13.5	14.5	12	11.5	12.7
BiLSTM-static	7	4.5	8	5.5	2	5.4
BiLSTM-dynamic	4	3	13	4	9.5	6.7
BiGRU-rand	14	11.5	12	13.5	13.5	12.9
BiGRU-static	7	8.5	3.5	3	5.5	5.5
BiGRU-dynamic	10	1	7	7.5	5.5	6.2
Stacked BiLSTM-rand	11	13.5	10	15	13.5	12.6
Stacked BiLSTM-static	9	11.5	5	11	5.5	8.4
Stacked BiLSTM-dynamic	2.5	4.5	14.5	5.5	5.5	6.5
Stacked BiGRU-rand	15	8.5	11	13.5	15	12.6
Stacked BiGRU-static	5	8.5	2	9	5.5	6
Stacked BiGRU-dynamic	7	2	6	7.5	9.5	6.4

5.2.7 Ensemble vs. Single Model

As expected, the ensemble models generally outperform the single-based models in almost all the classification tasks. As shown in Table 5.16, the ensemble model's static version provides better performance in 3 out of 5 datasets. The key to ensemble learning is that the candidate models need to be proven to work well on the given task. In this case, the 1D CNN and BiRNN are great models to combine for text classification. The result encourages us to experiment combining a potent model, such as TCN, with other existing good deep learning models in the future.

Table 5.16 The performance of ensemble vs. single models.

Model	Accuracy (%)				
	MR	*SUBJ*	*TREC*	*CR*	*MPQA*
1D CNN-rand *(baseline)*	77.6	92.05	89.8	80.4	86.4
1D CNN-static	79.0	92.51	92.2	81.4	**88.6**
1D CNN-dynamic	79.4	92.8	91.6	82.2	87.5
BiGRU-rand	77.2	92.2	89.0	80.1	86.1
BiGRU-static	79.5	92.3	91.8	82.4	88.1
BiGRU-dynamic	79.2	**93.0**	90.6	81.6	88.1
Ensemble CNN-GRU-rand	77.0	91.7	88.0	80.9	86.3
Ensemble CNN-GRU-static	**79.8**	92.7	**93.0**	**82.5**	88.4
Ensemble CNN-GRU-dynamic	79.4	92.6	89.6	82.4	88.0

Table 5.17 The average rank values (ARV) of ensemble vs. single model.

Model	Rank Average					ARV
	MR	*SUBJ*	*TREC*	*CR*	*MPQA*	
1D CNN-rand *(baseline)*	7	8	6	8	7	7.2
1D CNN-static	6	5	2	6	1	4
1D CNN-dynamic	3.5	2	4	4	6	3.9
BiGRU-rand	8	7	8	9	9	8.2
BiGRU-static	2	6	3	2.5	3.5	3.4
BiGRU-dynamic	5	1	5	5	3.5	3.9
Ensemble CNN-GRU-rand	9	9	9	7	8	8.4
Ensemble CNN-GRU-static	1	3	1	1	2	**1.6**
Ensemble CNN-GRU-dynamic	3.5	4	7	2.5	5	4.4

5.2.8 The Best Performing Models

Finally, Table 5.18 summarizes the best models in this series of experiments. We use the average accuracy margin in Figure 5.1 and the average rank values in Figure 5.2 to compare the top six performing models for classifying text. We can see that the static version of TCN and Ensemble models emerge as the best. Next, the TCN-dynamic follows as the best model joining the group as the top three. In the end, TCN and ensemble-based models dominate other configurations to perform text classification tasks, making them the best recommend architectures for future application and research.

Table 5.18 The top six deep learning models in this dissertation.

Top Six Deep Learning Models based on:	
the Average Accuracy Margin	the Average Rank Values
TCN-static	**Ensemble CNN-GRU-static**
Ensemble CNN-GRU-static	**TCN-static**
TCN-dynamic	**TCN-dynamic**
BiGRU-static	*1D CNN-static*
1D CNN-static	*BiGRU-static*
1D CNN-dynamic	*BiLSTM-static*

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This dissertation has demonstrated a comprehensive experiment focusing on building deep learning models using two different feature extractions on five text classification datasets. In conclusion, the followings are the essential insights from this project:

- When using the suitable feature extraction, such as word embedding, a deeper neural network can deliver a better final prediction;
- In the edRVFL model, sigmoid works as the best activation function for text classification task;
- To represent the text using BoW, binary proceeds as the best word scoring method, followed by freq, count, and TF-IDF.
- Any model built on top of word embedding causes the model to perform exceptionally well.
- Using a pre-trained word embedding such as Word2Vec can increase the model accuracy with a high margin.
- TCN is an excellent alternative to recurrent architecture and has been proven effective in classifying text data.
- The ensemble learning-based model can help make better predictions than a single model trained independently.
- TCN and Ensemble CNN-GRU models are the best performing algorithms we obtained in this series of text classification tasks.

6.2 Recommendation in Future Work

We recommend some suggestions for future experiments as follows:

- **An ensemble-based model with TCN.** Perform text classification tasks using TCN combined with other good models such as 1D CNN and BiGRU in ensemble-based learning to see if it can challenge the benchmarks even more.
- **Kernel size and filters**. Explore these two hyperparameters by extending the kernel sizes between 1 to 10 with more or fewer filters in CNN or TCN to see how it affects the model performance.
- **Deeper network.** Any neural network with more hidden layers typically will do better in any task. Explore the deeper version of CNN, RNN, and TCN to see how it affects the existing performance.
- **Vocabulary Size Reduction**. The BoW generally will perform better for small corpus/text data. Explore the text datasets by reducing the vocabulary size and see how it results.
- **Use GloVe and FastText**. Explore other pre-trained word embedding options such as GloVe and FastText with static and dynamic modes and compare the result to Word2Vec.

References

[1] Y. Kim, "Convolutional Neural Networks for Sentence Classification," *Association for Computational Linguistics*, October, 2014.

[2] S. bai, J. Kolter, and V. Koltun, "An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling", *arXiv*, April, 2018.

[3] K. Kowsari, M. Heidarysafa, D. E. Brown, K. J. Meimandi, and L. E. Barnes, "Random Multimodel Deep Learning for Classification", *arXiv*, April, 2018.

[4] R. Katuwal, P. N. Suganthan, M. Tanveer, "Random Vector Functional Link Neural Network based Ensemble Deep Learning", *arXiv*, June, 2019.

[5] B. Jang, M. Kim, G. Harerimana, S. Kang, and J. W. Kim, "BiLSTM Model to Increase Accuracy in Text Classification: Combining Word2vec CNN and Attention Mechanism", *Applied Sciences*, August, 2020.

[6] K. S. Jones, "A statistical interpretation of term specificity and its application in retrieval", *J. Doc*, 1972.

[7] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space", *arXiv*, 2013.

[8] Y. Bengio, P. Simard, P. Frasconi, "Learning long-term dependencies with gradient descent is difficult", *IEEE Trans. Neural Netw*. 1994.

[9] J. Chung, C. Gulcehre, K. Cho, Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling", *arXiv*, 2014.

[10] S. Hochreiter, J. Schmidhuber, “Long short-term memory”, *Neural Comput*, 1997.

[11] S. Lai, L. Xu, K. Liu, J. Zhao, "Recurrent Convolutional Neural Networks for Text Classification", In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, Austin, TX, USA, 25–30 January, 2015.

[12] Y. H. Pao, Y. Takefuji, "Functional-link net computing: theory, system architecture, and functionalities", *IEEE Computer 25*, 1992.

[13] L. Zhang, P. N. Suganthan, "A comprehensive evaluation of random vector functional link networks", *Information Sciences 367-368*, 2016.

[14] C. Henkel, "Temporal Convolutional Network", *Kaggle*, https://www.kaggle.com/christofhenkel/temporal-convolutional-network, February, 2021.

[15] S. Brahma, “Improved Sentence Modeling using Suffix Bidirectional LSTM”, *arXiv*, September, 2018.

[16] D. Cer, Y. Yang, S. Kong, N. Hua, N. Limtiaco, R. S. John, N. Constant, M. Guajardo-Cespedes, S. Yuan, C. Tar, Y. Sung, B. Strope, R. Kurzweil, “Universal Sentence Encoder”, *arXiv*, April, 2018.

[17] B. Pang, L. Lee, "Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales", In *Proceedings of ACL'05*, 2005.
[18] B. Pang, L. Lee, "A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts", In *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL'04)*, 2004.
[19] X. Li, D. Roth, "Learning question classifiers", In *Proceedings of COLING '02*, 2002.
[20] M. Hu, B. Liu, "Mining and summarizing customer reviews", In *Proceedings of KDD '04*, 2004.
[21] J. Wiebe, T. Wilson, and C. Cardie, "Annotating expressions of opinions and emotions in language", *Language Resources and Evaluation*, 39(2):165–210, 2005.
[22] AcademiaSinicaNLPLab, "sentiment_dataset", https://github.com/AcademiaSinicaNLPLab/sentiment_dataset, January, 2021.
[23] J. Pennington, R. Socher, C. D. Manning, "Glove: Global Vectors for Word Representation", In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar, 25–29 October 2014, Volume 14, pp. 1532–1543.
[24] P. Bojanowski, E. Grave, A. Joulin, T. Mikolov, "Enriching Word Vectors with Subword Information", *arXiv*, 2016.
[25] K. Kowsari, K. J. Meimandi, M. Heidarysafa, S. Mendu, L. Barnes, and D. Brown, "Text Classification Algorithms: A Survey", *arXiv*, May, 2020.
[26] M. Schuster, K.K. Paliwal, "Bidirectional recurrent neural networks", *IEEE*, November, 1997.

Appendix A

Python Program for Feedforward Models

```python
"""
Title              : Python Program for Feedforward Models
Dataset            : CR (as an example)
Feature Extration: Bag-of-Words
Author             : Diardano Raihan
"""

##=========================Import Libraries========================#
import re
import numpy as np
import pandas as pd
import tensorflow as tf
from string import punctuation
from collections import Counter
import matplotlib.pyplot as plt
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from sklearn.model_selection import KFold
from tensorflow.keras.preprocessing.text import Tokenizer

#==============================Step 1==============================#
# Load the dataset
corpus = pd.read_pickle('../../0_data/CR/CR.pkl')
#==============================Step 2==============================#
# Define functions needed for:
# 1. Cleaning the text and turning into tokens
# 2. Creating a vocabulary list for BoW
stopwords = stopwords.words('english')
stemmer = PorterStemmer()
def clean_doc(doc):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(punctuation))
    # remove punctuation from each word
    tokens = [re_punc.sub('', w) for w in tokens]
    # filter out stop words
    tokens = [w for w in tokens if not w in stopwords]
    # filter out short tokens
    tokens = [word for word in tokens if len(word) >= 1]
    # Stem the token
    tokens = [stemmer.stem(token) for token in tokens]
    return tokens
```

```python
vocab = Counter()

def add_doc_to_vocab(docs, vocab):

    for doc in docs:
        tokens = clean_doc(doc)
        vocab.update(tokens)
    return vocab

def doc_to_line(doc):
    tokens = clean_doc(doc)
    # filter by vocab
    tokens = [token for token in tokens if token in vocab]
    line = ' '.join([token for token in tokens])
    return line

def clean_docs(docs):
    lines = []
    for doc in docs:
        line = doc_to_line(doc)
        lines.append(line)
    return lines

def create_tokenizer(sentence):
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer

#==============================Step 3============================#
# Define the feedforward model SNN-a
def train_mlp_1(train_x, train_y, batch_size = 50, epochs = 10,
                verbose =2):

    n_words = train_x.shape[1]

    model = tf.keras.models.Sequential([
        tf.keras.layers.Dense(units=50,
                              activation='ReLU',
                              input_shape=(n_words,)),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense( units=1, activation='sigmoid')
    ])

    model.compile(loss = 'binary_crossentropy',
                  optimizer = 'adam',
                  metrics = ['accuracy'])
    model.fit(train_x, train_y, batch_size, epochs, verbose)
    return model

# Define the feedforward model SNN-b
def train_mlp_2(train_x, train_y, batch_size = 50, epochs = 10,
                verbose =2):

    n_words = train_x.shape[1]
```

```
    model = tf.keras.models.Sequential([
        tf.keras.layers.Dense( units=100, activation='ReLU',
                              input_shape=(n_words,)),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense( units=1, activation='sigmoid')
    ])

    model.compile(loss = 'binary_crossentropy',
                  optimizer = 'adam',
                  metrics = ['accuracy'])
    model.fit(train_x, train_y, batch_size, epochs, verbose)
    return model

# Define the feedforward model SNN-b
def train_mlp_3(train_x, train_y, batch_size = 50, epochs = 10,
                verbose =2):

    n_words = train_x.shape[1]

    model = tf.keras.models.Sequential([
        tf.keras.layers.Dense( units=100, activation='ReLU',
                              input_shape=(n_words,)),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense( units=50, activation='ReLU'),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense( units=1, activation='sigmoid')
    ])

    model.compile(loss = 'binary_crossentropy',
                  optimizer = 'adam',
                  metrics = ['accuracy'])
    model.fit(train_x, train_y, batch_size, epochs, verbose)
    return model

# Define the early stopping callbacks
callbacks = tf.keras.callbacks.EarlyStopping(
            monitor='val_accuracy',
            min_delta=0,
            patience=5, verbose=2,
            mode='auto',
            restore_best_weights=True)

#===============================Step 4============================#
# Begin the training process with BoW four word scoring options
# prepare cross validation with 10 splits and shuffle = True
kfold = KFold(10, True)

# Separate the sentences and the labels
sentences, labels = list(corpus.sentence), list(corpus.label)

# Run Experiment of 4 different modes of BoW word scoring
modes = ['binary', 'count', 'tfidf', 'freq']
results_1 = pd.DataFrame()
results_2 = pd.DataFrame()
results_3 = pd.DataFrame()
```

```python
for mode in modes:
    print('mode: ', mode)
    acc_list_1 = []
    acc_list_2 = []
    acc_list_3 = []


    # kfold.split() will return set indices for each split
    for train, test in kfold.split(sentences):
        # Instantiate a vocab object
        vocab = Counter()

        train_x, test_x = [], []
        train_y, test_y = [], []

        for i in train:
            train_x.append(sentences[i])
            train_y.append(labels[i])

        for i in test:
            test_x.append(sentences[i])
            test_y.append(labels[i])

        # Turn the labels into a numpy array
        train_y = np.array(train_y)
        test_y = np.array(test_y)

        # Define a vocabulary for each fold
        vocab = add_doc_to_vocab(train_x, vocab)
        # print('The number of vocab: ', len(vocab))

        # Clean the sentences
        train_x = clean_docs(train_x, vocab)
        test_x = clean_docs(test_x, vocab)

        # encode data using freq mode
        Xtrain, Xtest = prepare_data(train_x, test_x, mode)

        # train the model
        model_1 = train_mlp_1(Xtrain, train_y, Xtest, test_y,
                              verbose=1)
        model_2 = train_mlp_2(Xtrain, train_y, Xtest, test_y,
                              verbose=1)
        model_3 = train_mlp_3(Xtrain, train_y, Xtest, test_y,
                              verbose=1)


        # evaluate the model
        loss_1, acc_1 = model_1.evaluate(Xtest, test_y, verbose=0)
        loss_2, acc_2 = model_2.evaluate(Xtest, test_y, verbose=0)
        loss_3, acc_3 = model_3.evaluate(Xtest, test_y, verbose=0)

        acc_list_1.append(acc_1)
        acc_list_2.append(acc_2)
        acc_list_3.append(acc_3)
```

```
    results_1[mode] = acc_list_1
    results_2[mode] = acc_list_2
    results_3[mode] = acc_list_3

print(results_1)
print(results_2)
print(results_3)

#===============================Step 5=============================#
# Save the dataframe into excel file
results_1.to_excel('BoW_SNN_a.xlsx', sheet_name='model_1')
results_2.to_excel('BoW_SNN_b.xlsx', sheet_name='model_2')
results_3.to_excel('BoW_SNN_c.xlsx', sheet_name='model_3')
#==================================================================#
```

```python
"""
Title            : Python Program for Feedforward Models
Dataset          : CR (as an example)
Feature Extration: Average Word2Vec
Author           : Diardano Raihan
"""

##=========================Import Libraries======================#
import re
import numpy as np
import pandas as pd
import tensorflow as tf
from string import punctuation
from collections import Counter
import matplotlib.pyplot as plt
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from gensim.models import KeyedVectors
from sklearn.model_selection import KFold
from tensorflow.keras.preprocessing.text import Tokenizer

#==============================Step 1============================#
# Load the dataset
corpus = pd.read_pickle('../../0_data/CR/CR.pkl')
# Load the Word2Vec
word2vec = KeyedVectors.load_word2vec_format(
            '../GoogleNews-vectors-negative300.bin',
            binary=True)

#==============================Step 2============================#
# Define functions needed to:
# 1. Clean the text
# 2. Convert the text into vectors based on Word2Vec

def clean_doc(sentences, word_index):
    clean_sentences = []
    for sentence in sentences:
        sentence = sentence.lower().split()
        clean_word = []
        for word in sentence:
            if word in word_index:
                clean_word.append(word)
        clean_sentence = ' '.join(clean_word)
        clean_sentences.append(clean_sentence)
    return clean_sentences

def sentence_to_avg(sentence, word_to_vec_map):

    # Split sentence into list of lower case words
    words = (sentence.lower()).split()

    # Initialize the average word vector
    avg = np.zeros(word2vec.word_vec('i').shape)
```

```
    # Average the word vectors
    total = 0
    count = 0
    for w in words:
        if w in word_to_vec_map:
            total += word_to_vec_map.word_vec(w)
            count += 1

    if count!= 0:
        avg = total/count
    else:
        avg
    return avg

# Encode Sentence into Word2Vec Representation
def encoded_sentences(sentences):

    encoded_sentences = []

    for sentence in sentences:

        encoded_sentence = sentence_to_avg(sentence, word2vec)
        encoded_sentences.append(encoded_sentence)

    encoded_sentences = np.array(encoded_sentences)
    return encoded_sentences

#===============================Step 3=============================#
# Model Definition

# Define the feedforward model SNN-a
def define_model(input_length=300):

    model = tf.keras.models.Sequential([
        tf.keras.layers.Dense(units=50, activation='ReLU',
                              input_shape=(input_length,)),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense( units=1, activation='sigmoid')
    ])

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

# Define the feedforward model SNN-b
def define_model_2(input_length=300):

    model = tf.keras.models.Sequential([
        tf.keras.layers.Dense(units=100,
                              activation='ReLU',
                              input_shape=(input_length,)),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense( units=1, activation='sigmoid')
    ])
```

```
    model.compile(loss = 'binary_crossentropy',
                  optimizer = 'adam',
                  metrics = ['accuracy'])
    return model

# Define the feedforward model SNN-c
def define_model_3(input_length=300):

    model = tf.keras.models.Sequential([
        tf.keras.layers.Dense(units=100,
                              activation='ReLU',
                              input_shape=(input_length,)),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense( units=50, activation='ReLU'),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense( units=1, activation='sigmoid')
    ])

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

# Define the early stopping callbacks
callbacks = tf.keras.callbacks.EarlyStopping(
            monitor='val_accuracy',
            min_delta=0,
            patience=10, verbose=2,
            mode='auto',
            restore_best_weights=True)

#==============================Step 4============================#
# Begin the training process with Average Word2Vec

# Parameter Initialization
oov_tok = "<UNK>"
columns = ['acc1', 'acc2', 'acc3', 'acc4', 'acc5', 'acc6',
           'acc7', 'acc8', 'acc9', 'acc10', 'AVG']

record_1 = pd.DataFrame(columns = columns)
record_2 = pd.DataFrame(columns = columns)
record_3 = pd.DataFrame(columns = columns)

sentences, labels = list(corpus.sentence), list(corpus.label)

# prepare cross validation with 10 splits and shuffle = True
kfold = KFold(10, True)

exp=0
acc_list_1 = []
acc_list_2 = []
acc_list_3 = []
```

```python
# kfold.split() will return set indices for each split
for train, test in kfold.split(sentences):

    exp+=1
    print('Training {}: '.format(exp))

    train_x, test_x = [], []
    train_y, test_y = [], []

    for i in train:
        train_x.append(sentences[i])
        train_y.append(labels[i])

    for i in test:
        test_x.append(sentences[i])
        test_y.append(labels[i])

    # Turn the data into a numpy array
    train_y = np.array(train_y)
    test_y = np.array(test_y)

    # Define the word_index
    tokenizer = Tokenizer(oov_token=oov_tok)
    tokenizer.fit_on_texts(train_x)
    word_index = tokenizer.word_index

    # Clean the sentences
    Xtrain = clean_doc(train_x, word_index)
    Xtest = clean_doc(test_x, word_index)

    # Encode the sentences into average Word2Vec representation
    Xtrain = encoded_sentences(Xtrain)
    Xtest = encoded_sentences(Xtest)

    # Define the input shape
    model_1 = define_model_1(Xtrain.shape[1])
    model_2 = define_model_2(Xtrain.shape[1])
    model_3 = define_model_3(Xtrain.shape[1])

    # Train the model
    model_1.fit(Xtrain, train_y, batch_size=32,
                epochs=40, verbose=1,
                callbacks=[callbacks],
                validation_data=(Xtest, test_y))

    model_2.fit(Xtrain, train_y, batch_size=32,
                epochs=40, verbose=1,
                callbacks=[callbacks],
                validation_data=(Xtest, test_y))

    model_3.fit(Xtrain, train_y, batch_size=32,
                epochs=40, verbose=1,
                callbacks=[callbacks],
                validation_data=(Xtest, test_y))
```

```
    # evaluate the model
    loss_1, acc_1 = model_1.evaluate(Xtest, test_y, verbose=0)
    loss_2, acc_2 = model_2.evaluate(Xtest, test_y, verbose=0)
    loss_3, acc_3 = model_3.evaluate(Xtest, test_y, verbose=0)


    acc_list_1.append(acc_1)
    acc_list_2.append(acc_2)
    acc_list_3.append(acc_3)

mean_acc_1 = np.array(acc_list_1).mean()
mean_acc_2 = np.array(acc_list_2).mean()
mean_acc_3 = np.array(acc_list_3).mean()

entries_1 = acc_list_1 + [mean_acc_1]
entries_2 = acc_list_2 + [mean_acc_2]
entries_3 = acc_list_1 + [mean_acc_3]

temp = pd.DataFrame([entries_1], columns=columns)
record_1 = record_1.append(temp, ignore_index=True)
temp = pd.DataFrame([entries_2], columns=columns)
record_2 = record_2.append(temp, ignore_index=True)
temp = pd.DataFrame([entries_3], columns=columns)
record_3 = record_3.append(temp, ignore_index=True)

print(record_1)
print(record_2)
print(record_3)

#==============================Step 5============================#
# Save the dataframe into excel file
record_1.to_excel('WE_SNN_a.xlsx', sheet_name='model_1')
record_2.to_excel('WE_SNN_b.xlsx', sheet_name='model_2')
record_3.to_excel('WE_SNN_c.xlsx', sheet_name='model_3')
#================================================================#
```

Appendix B

Python Program for CNN Models

```python
"""
Title             : Python Program for CNN Models
Dataset           : CR (as an example)
Feature Extration: Random, Static, Dynamic Word2Vec
Author            : Diardano Raihan
"""

##=========================Import Libraries======================#
import re
import numpy as np
import pandas as pd
import tensorflow as tf
from string import punctuation
from collections import Counter
import matplotlib.pyplot as plt
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from gensim.models import KeyedVectors
from sklearn.model_selection import KFold
from tensorflow.keras import regularizers
from tensorflow.keras.constraints import MaxNorm
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

#==============================Step 1============================#
# Load the dataset
corpus = pd.read_pickle('../../0_data/CR/CR.pkl')
# Load the Word2Vec
word2vec = KeyedVectors.load_word2vec_format(
            '../GoogleNews-vectors-negative300.bin',
            binary=True)

#==============================Step 2============================#
# Defined all the functions needed as Text Preprocessing steps

# Define a function to compute the max length of sequence
def max_length(sequences):
    '''
    input:
        sequences: a 2D list of integer sequences
    output:
        max_length: the max length of the sequences
    '''
    max_length = 0
    for i, seq in enumerate(sequences):
        length = len(seq)
```

```python
        if max_length < length:
            max_length = length
    return max_length

# Calculated the statistics of Word2Vec
emb_mean = word2vec.vectors.mean()
emb_std = word2vec.vectors.std()

# Map the Word2Vec into a matrix for embedding weights
def pretrained_embedding_matrix(word_to_vec_map,
                                word_to_index,
                                emb_mean, emb_std):

    np.random.seed(2021)

    # adding 1 to fit Keras embedding (requirement)
    vocab_size = len(word_to_index) + 1
    # define dimensionality of your pre-trained word vectors
    emb_dim = word_to_vec_map.word_vec('handsome').shape[0]

    # initialize the matrix with generic normal distribution
    embed_matrix = np.random.normal(emb_mean,
                                    emb_std,
                                    (vocab_size, emb_dim))

    # Set each row "idx" of the embedding matrix to be
    # the word vector representation of the idx'th
    # word of the vocabulary
    for word, idx in word_to_index.items():
        if word in word_to_vec_map:

            embed_matrix[idx] = word_to_vec_map.get_vector(word)

    return embed_matrix

#==============================Step 3============================#
# CNN-rand
def define_model(filters = 100, kernel_size = 3,
                 activation='ReLU', input_dim = None,
                 output_dim=300, max_length = None ):

    model = tf.keras.models.Sequential([
        tf.keras.layers.Embedding(input_dim=vocab_size,
                                  output_dim=output_dim,
                                  input_length=max_length,
                                  input_shape=(max_length, )),

        tf.keras.layers.Conv1D(filters=filters,
                               kernel_size = kernel_size,
                               activation = activation,
                               # set 'axis' value to the first and
                               # second axis of conv1D weights
                               # (rows, cols)
                               kernel_constraint = MaxNorm(
                                   max_value=3,
```

```
                                 axis=[0,1])),

        tf.keras.layers.MaxPool1D(2),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(10, activation=activation,
                              # set axis to 0 to constrain
                              # each weight vector of length
                              # (input_dim,) in dense layer
                              kernel_constraint = MaxNorm(
                                  max_value=3, axis=0)),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(units=1, activation='sigmoid')
    ])

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

# CNN-static
def define_model_2(filters = 100, kernel_size = 3,
                   activation='ReLU', input_dim = None,
                   output_dim=300, max_length = None,
                   emb_matrix = None):

    model = tf.keras.models.Sequential([
        tf.keras.layers.Embedding(input_dim=input_dim,
                                  output_dim=output_dim,
                                  input_length=max_length,
                                  input_shape=(max_length, ),
                                  # Assign the embedding weight
                                  # with word2vec embedding marix
                                  weights = [emb_matrix],
                                  # Set the weight to be not
                                  # trainable (static)
                                  trainable = False),

        tf.keras.layers.Conv1D(filters=filters,
                               kernel_size = kernel_size,
                               activation = activation,
                               # set 'axis' value to the first and
                               # second axis of conv1D weights
                               # (rows, cols)
                               kernel_constraint= MaxNorm(
                                   max_value=3, axis=[0,1])),

        tf.keras.layers.MaxPool1D(2),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(10, activation=activation,
                              # set axis to 0 to constrain
                              # each weight vector of length
                              # (input_dim,) in dense layer
                              kernel_constraint = MaxNorm(
```

```
                                max_value=3, axis=0)),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(units=1, activation='sigmoid')
    ])

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

    return model

# CNN-dynamic
def define_model_3(filters = 100, kernel_size = 3,
                   activation='ReLU', input_dim = None,
                   output_dim=300, max_length = None,
                   emb_matrix = None):

    model = tf.keras.models.Sequential([
        tf.keras.layers.Embedding(input_dim=input_dim,
                                  output_dim=output_dim,
                                  input_length=max_length,
                                  input_shape=(max_length, ),
                                  # Assign the embedding weight
                                  # with word2vec embedding marix
                                  weights = [emb_matrix],
                                  # Set the weight to be not
                                  # trainable (static)
                                  trainable = True),

        tf.keras.layers.Conv1D(filters=filters,
                               kernel_size = kernel_size,
                               activation = activation,
                               # set 'axis' value to the first and
                               # second axis of conv1D weights
                               # (rows, cols)
                               kernel_constraint= MaxNorm(
                                   max_value=3, axis=[0,1])),

        tf.keras.layers.MaxPool1D(2),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(10, activation=activation,
                              # set axis to 0 to constrain
                              # each weight vector of length
                              # (input_dim,) in dense layer
                              kernel_constraint = MaxNorm(
                                  max_value=3, axis=0)),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(units=1, activation='sigmoid')
    ])
    model.compile(loss = 'binary_crossentropy',
                  optimizer = 'adam',
                  metrics = ['accuracy'])

    return model
```

```python
# Define the early stopping callbacks
callbacks = tf.keras.callbacks.EarlyStopping(
            monitor='val_accuracy',
            min_delta=0,
            patience=10, verbose=2,
            mode='auto',
            restore_best_weights=True)

#===============================Step 4=============================#
# Begin the training process with random/static/dynamic Word2Vec
# Parameter Initialization
trunc_type='post'
padding_type='post'
oov_tok = "<UNK>"
activations = ['ReLU', 'tanh']
filters = 100
kernel_sizes = [1, 2, 3, 4, 5, 6]
emb_mean = emb_mean
emb_std = emb_std

columns = ['Activation', 'Filters', 'acc1', 'acc2', 'acc3',
           'acc4', 'acc5', 'acc6', 'acc7', 'acc8', 'acc9',
           'acc10', 'AVG']

record_1 = pd.DataFrame(columns = columns)
record_2 = pd.DataFrame(columns = columns)
record_3 = pd.DataFrame(columns = columns)

# prepare cross validation with 10 splits and shuffle = True
kfold = KFold(10, True)

# Separate the sentences and the labels
sentences, labels = list(corpus.sentence), list(corpus.label)

for activation in activations:
    for kernel_size in kernel_sizes:
        # kfold.split() will return set indices for each split
        acc_list_1 = []
        acc_list_2 = []
        acc_list_3 = []

        for train, test in kfold.split(sentences):

            train_x, test_x = [], []
            train_y, test_y = [], []

            for i in train:
                train_x.append(sentences[i])
                train_y.append(labels[i])


            for i in test:
                test_x.append(sentences[i])
```

```
    test_y.append(labels[i])

# Turn the labels into a numpy array
train_y = np.array(train_y)
test_y = np.array(test_y)

# encode data using
# Cleaning and Tokenization
tokenizer = Tokenizer(oov_token=oov_tok)
tokenizer.fit_on_texts(train_x)

# Turn the text into sequence
training_sequences = tokenizer.texts_to_sequences(
                     train_x)
test_sequences = tokenizer.texts_to_sequences(
                 test_x)

max_len = max_length(training_sequences)

# Pad the sequence to have the same size
Xtrain = pad_sequences(training_sequences,
                       maxlen=max_len,
                       padding=padding_type,
                       truncating=trunc_type)
Xtest = pad_sequences(test_sequences,
                      maxlen=max_len,
                      padding=padding_type,
                      truncating=trunc_type)

word_index = tokenizer.word_index
vocab_size = len(word_index)+1

emb_matrix = pretrained_embedding_matrix(word2vec,
                                         word_index,
                                         emb_mean,
                                         emb_std)
# Define the models to train
model_1 = define_model(filters,
                       kernel_size,
                       activation,
                       input_dim=vocab_size,
                       max_length=max_len)

model_2 = define_model_2(filters,
                         kernel_size,
                         activation,
                         input_dim=vocab_size,
                         max_length=max_len,
                         emb_matrix=emb_matrix)

model_3 = define_model_3(filters,
                         kernel_size,
                         activation,
                         input_dim=vocab_size,
                         max_length=max_len,
```

```
                                    emb_matrix=emb_matrix)

            # Train the models
            model_1.fit(Xtrain, train_y, batch_size=50,
                        epochs=100, verbose=1,
                        callbacks=[callbacks],
                        validation_data=(Xtest, test_y))

            model_2.fit(Xtrain, train_y, batch_size=50,
                        epochs=100, verbose=1,
                        callbacks=[callbacks],
                        validation_data=(Xtest, test_y))

            model_3.fit(Xtrain, train_y, batch_size=50,
                        epochs=100, verbose=1,
                        callbacks=[callbacks],
                        validation_data=(Xtest, test_y))

            # evaluate the model
            loss_1, acc_1 = model_1.evaluate(Xtest, test_y)
            loss_2, acc_2 = model_2.evaluate(Xtest, test_y)
            loss_3, acc_3 = model_3.evaluate(Xtest, test_y)

            acc_list_1.append(acc*100)
            acc_list_2.append(acc*100)
            acc_list_3.append(acc*100)

        mean_acc_1 = np.array(acc_list_1).mean()
        mean_acc_2 = np.array(acc_list_2).mean()
        mean_acc_3 = np.array(acc_list_3).mean()

        parameters = [activation, kernel_size]
        entries_1 = parameters + acc_list + [mean_acc_1]
        entries_2 = parameters + acc_list + [mean_acc_2]
        entries_3 = parameters + acc_list + [mean_acc_3]

        temp = pd.DataFrame([entries_1], columns=columns)
        record_1 = record_1.append(temp, ignore_index=True)
        temp = pd.DataFrame([entries_2], columns=columns)
        record_2 = record_2.append(temp, ignore_index=True)
        temp = pd.DataFrame([entries_3], columns=columns)
        record_3 = record_3.append(temp, ignore_index=True)

        print(record_1)
        print(record_2)
        print(record_3)

#===============================Step 5=============================#
# Save the dataframe into excel file
record_1.to_excel('WE_CNN_1.xlsx', sheet_name='random')
record_2.to_excel('WE_CNN_2.xlsx', sheet_name='static')
record_3.to_excel('WE_CNN_3.xlsx', sheet_name='dynamic')
#==================================================================#
```

Appendix C

Python Program for TCN Models

```python
"""
Title             : Python Program for TCN Models
Dataset           : CR (as an example)
Feature Extration: Random, Static, Dynamic Word2Vec
Author            : Diardano Raihan (inspired by: christofhenkel)
https://www.kaggle.com/christofhenkel/temporal-convolutional-network
"""

##=========================Import Libraries======================#
import re
import numpy as np
import pandas as pd
import tensorflow as tf
from string import punctuation
from collections import Counter
import matplotlib.pyplot as plt
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from gensim.models import KeyedVectors
from sklearn.model_selection import KFold
from tensorflow.keras import regularizers
from tensorflow.keras.constraints import MaxNorm
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

from tcn import TCN, tcn_full_summary
from tensorflow.keras.layers import Input, Embedding
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.layers import SpatialDropout1D
from tensorflow.keras.layers import concatenate
from tensorflow.keras.layers import GlobalAveragePooling1D
from tensorflow.keras.layers import GlobalMaxPooling1D
from tensorflow.keras.models import Model

#==============================Step 1============================#
# Load the dataset
corpus = pd.read_pickle('../../0_data/CR/CR.pkl')
# Load the Word2Vec
word2vec = KeyedVectors.load_word2vec_format(
            '../GoogleNews-vectors-negative300.bin',
            binary=True)

#==============================Step 2============================#
# Defined all the functions needed as Text Preprocessing steps

# Define a function to compute the max length of sequence
```

```python
def max_length(sequences):
    '''
    input:
        sequences: a 2D list of integer sequences
    output:
        max_length: the max length of the sequences
    '''
    max_length = 0
    for i, seq in enumerate(sequences):
        length = len(seq)
        if max_length < length:
            max_length = length
    return max_length

# Calculated the statistics of Word2Vec
emb_mean = word2vec.vectors.mean()
emb_std = word2vec.vectors.std()

# Map the Word2Vec into a matrix for embedding weights
def pretrained_embedding_matrix(word_to_vec_map,
                                word_to_index,
                                emb_mean, emb_std):

    np.random.seed(2021)

    # adding 1 to fit Keras embedding (requirement)
    vocab_size = len(word_to_index) + 1
    # define dimensionality of your pre-trained word vectors
    emb_dim = word_to_vec_map.word_vec('handsome').shape[0]

    # initialize the matrix with generic normal distribution
    embed_matrix = np.random.normal(emb_mean,
                                    emb_std,
                                    (vocab_size, emb_dim))

    # Set each row "idx" of the embedding matrix to be
    # the word vector representation of the idx'th
    # word of the vocabulary
    for word, idx in word_to_index.items():
        if word in word_to_vec_map:

            embed_matrix[idx] = word_to_vec_map.get_vector(word)

    return embed_matrix

#==============================Step 3=============================#
# TCN-rand
def define_model(kernel_size = 3, activation='ReLU',
                 input_dim = None, output_dim=300,
                 max_length = None ):

    inp = Input( shape=(max_length,))
    x = Embedding(input_dim=input_dim,
                  output_dim=output_dim,
                  input_length=max_length)(inp)
```

```python
    x = SpatialDropout1D(0.1)(x)

    x = TCN(128,dilations = [1, 2, 4],
            kernel_size = kernel_size,
            return_sequences=True,
            activation = activation,
            name = 'tcn1')(x)

    x = TCN(64,dilations = [1, 2, 4],
            kernel_size = kernel_size,
            return_sequences=True,
            activation = activation,
            name = 'tcn2')(x)

    avg_pool = GlobalAveragePooling1D()(x)
    max_pool = GlobalMaxPooling1D()(x)

    conc = concatenate([avg_pool, max_pool])
    conc = Dense(16, activation="ReLU")(conc)
    conc = Dropout(0.1)(conc)
    outp = Dense(1, activation="sigmoid")(conc)

    model = Model(inputs=inp, outputs=outp)
    model.compile(loss = 'binary_crossentropy',
                  optimizer = 'adam',
                  metrics = ['accuracy'])

    return model

# TCN-static
def define_model_2(kernel_size = 3, activation='ReLU',
                   input_dim = None,
                   output_dim=300, max_length = None,
                   emb_matrix = None):

    inp = Input( shape=(max_length,))
    x = Embedding(input_dim=input_dim,
                  output_dim=output_dim,
                  input_length=max_length,
                  # Assign the embedding weight with
                  # word2vec embedding marix
                  weights = [emb_matrix],
                  # Set the weight to be not trainable
                  # (static)
                  trainable = False)(inp)

    x = SpatialDropout1D(0.1)(x)

    x = TCN(128,dilations = [1, 2, 4],
            kernel_size = kernel_size,
            return_sequences=True,
            activation = activation,
            name = 'tcn1')(x)
    x = TCN(64,dilations = [1, 2, 4],
```

```
            kernel_size = kernel_size,
            return_sequences=True,
            activation = activation,
            name = 'tcn2')(x)
    avg_pool = GlobalAveragePooling1D()(x)
    max_pool = GlobalMaxPooling1D()(x)
    conc = concatenate([avg_pool, max_pool])
    conc = Dense(16, activation="ReLU")(conc)
    conc = Dropout(0.1)(conc)
    outp = Dense(1, activation="sigmoid")(conc)

    model = Model(inputs=inp, outputs=outp)
    model.compile( loss = 'binary_crossentropy',
                  optimizer = 'adam',
                  metrics = ['accuracy'])

    return model

# TCN-dynamic
def define_model_3(kernel_size = 3, activation='ReLU',
                   input_dim = None,
                   output_dim=300, max_length = None,
                   emb_matrix = None):

    inp = Input( shape=(max_length,))
    x = Embedding(input_dim=input_dim,
                  output_dim=output_dim,
                  input_length=max_length,
                  # Assign the embedding weight with
                  # word2vec embedding marix
                  weights = [emb_matrix],
                  # Set the weight to be not trainable
                  # (static)
                  trainable = True)(inp)

    x = SpatialDropout1D(0.1)(x)

    x = TCN(128,dilations = [1, 2, 4],
            kernel_size = kernel_size,
            return_sequences=True,
            activation = activation,
            name = 'tcn1')(x)
    x = TCN(64,dilations = [1, 2, 4],
            kernel_size = kernel_size,
            return_sequences=True,
            activation = activation,
            name = 'tcn2')(x)

    avg_pool = GlobalAveragePooling1D()(x)
    max_pool = GlobalMaxPooling1D()(x)

    conc = concatenate([avg_pool, max_pool])
    conc = Dense(16, activation="ReLU")(conc)
    conc = Dropout(0.1)(conc)
    outp = Dense(1, activation="sigmoid")(conc)
```

```python
    model = Model(inputs=inp, outputs=outp)
    model.compile( loss = 'binary_crossentropy',
                  optimizer = 'adam',
                  metrics = ['accuracy'])

    return model

# Define the early stopping callbacks
callbacks = tf.keras.callbacks.EarlyStopping(
            monitor='val_accuracy',
            min_delta=0,
            patience=10, verbose=2,
            mode='auto',
            restore_best_weights=True)

#===============================Step 3=============================#
# Begin the training process with random/static/dynamic Word2Vec
# Parameter Initialization
trunc_type='post'
padding_type='post'
oov_tok = "<UNK>"
activations = ['ReLU', 'tanh']
filters = 100
kernel_sizes = [1, 2, 3, 4, 5, 6]
emb_mean = emb_mean
emb_std = emb_std

columns = ['Activation', 'Filters', 'acc1', 'acc2', 'acc3',
           'acc4', 'acc5', 'acc6', 'acc7', 'acc8', 'acc9',
           'acc10', 'AVG']
record_1 = pd.DataFrame(columns = columns)
record_2 = pd.DataFrame(columns = columns)
record_3 = pd.DataFrame(columns = columns)

# prepare cross validation with 10 splits and shuffle = True
kfold = KFold(10, True)

# Separate the sentences and the labels
sentences, labels = list(corpus.sentence), list(corpus.label)

for activation in activations:
    for kernel_size in kernel_sizes:
        # kfold.split() will return set indices for each split
        acc_list_1 = []
        acc_list_2 = []
        acc_list_3 = []

        for train, test in kfold.split(sentences):

            train_x, test_x = [], []
            train_y, test_y = [], []

            for i in train:
                train_x.append(sentences[i])
```

```
    train_y.append(labels[i])

for i in test:
    test_x.append(sentences[i])
    test_y.append(labels[i])

# Turn the labels into a numpy array
train_y = np.array(train_y)
test_y = np.array(test_y)

# encode data using
# Cleaning and Tokenization
tokenizer = Tokenizer(oov_token=oov_tok)
tokenizer.fit_on_texts(train_x)

# Turn the text into sequence
training_sequences = tokenizer.texts_to_sequences(
                     train_x)
test_sequences = tokenizer.texts_to_sequences(
                 test_x)

max_len = max_length(training_sequences)

# Pad the sequence to have the same size
Xtrain = pad_sequences(training_sequences,
                       maxlen=max_len,
                       padding=padding_type,
                       truncating=trunc_type)
Xtest = pad_sequences(test_sequences,
                      maxlen=max_len,
                      padding=padding_type,
                      truncating=trunc_type)

word_index = tokenizer.word_index
vocab_size = len(word_index)+1


emb_matrix = pretrained_embedding_matrix(word2vec,
                                         word_index,
                                         emb_mean,
                                         emb_std)
# Define the models to train
model_1 = define_model(filters,
                       kernel_size,
                       activation,
                       input_dim=vocab_size,
                       max_length=max_len)

model_2 = define_model_2(filters,
                         kernel_size,
                         activation,
                         input_dim=vocab_size,
                         max_length=max_len,
                         emb_matrix=emb_matrix)
```

```
            model_3 = define_model_3(filters,
                                     kernel_size,
                                     activation,
                                     input_dim=vocab_size,
                                     max_length=max_len,
                                     emb_matrix=emb_matrix)
            # Train the models
            model_1.fit(Xtrain, train_y, batch_size=50,
                        epochs=100, verbose=1,
                        callbacks=[callbacks],
                        validation_data=(Xtest, test_y))
            model_2.fit(Xtrain, train_y, batch_size=50,
                        epochs=100, verbose=1,
                        callbacks=[callbacks],
                        validation_data=(Xtest, test_y))
            model_3.fit(Xtrain, train_y, batch_size=50,
                        epochs=100, verbose=1,
                      callbacks=[callbacks],
                        validation_data=(Xtest, test_y))

            # evaluate the model
            loss_1, acc_1 = model_1.evaluate(Xtest, test_y)
            loss_2, acc_2 = model_2.evaluate(Xtest, test_y)
            loss_3, acc_3 = model_3.evaluate(Xtest, test_y)

            acc_list_1.append(acc*100)
            acc_list_2.append(acc*100)
            acc_list_3.append(acc*100)

        mean_acc_1 = np.array(acc_list_1).mean()
        mean_acc_2 = np.array(acc_list_2).mean()
        mean_acc_3 = np.array(acc_list_3).mean()

        parameters = [activation, kernel_size]
        entries_1 = parameters + acc_list + [mean_acc_1]
        entries_2 = parameters + acc_list + [mean_acc_2]
        entries_3 = parameters + acc_list + [mean_acc_3]

        temp = pd.DataFrame([entries_1], columns=columns)
        record_1 = record_1.append(temp, ignore_index=True)
        temp = pd.DataFrame([entries_2], columns=columns)
        record_2 = record_2.append(temp, ignore_index=True)
        temp = pd.DataFrame([entries_3], columns=columns)
        record_3 = record_3.append(temp, ignore_index=True)

        print(record_1)
        print(record_2)
        print(record_3)

#================================Step 5=============================#
# Save the dataframe into excel file
record_1.to_excel('WE_TCN_1.xlsx', sheet_name='random')
record_2.to_excel('WE_TCN_2.xlsx', sheet_name='static')
record_3.to_excel('WE_TCN_3.xlsx', sheet_name='dynamic')
#==================================================================#
```

Appendix D

Python Program for BiGRU/BiLSTM Models

```
"""
Title             : Python Program for BiGRU/BiLSTM Models
Dataset           : CR (as an example)
Feature Extration: Random, Static, Dynamic Word2Vec
Author            : Diardano Raihan
"""

##=========================Import Libraries=======================#
import re
import numpy as np
import pandas as pd
import tensorflow as tf
from string import punctuation
from collections import Counter
import matplotlib.pyplot as plt
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from gensim.models import KeyedVectors
from sklearn.model_selection import KFold
from tensorflow.keras import regularizers
from tensorflow.keras.constraints import MaxNorm
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

#==============================Step 1============================#
# Load the dataset
corpus = pd.read_pickle('../../0_data/CR/CR.pkl')
# Load the Word2Vec
word2vec = KeyedVectors.load_word2vec_format(
            '../GoogleNews-vectors-negative300.bin',
            binary=True)

#==============================Step 2============================#
# Defined all the functions needed as Text Preprocessing steps

# Define a function to compute the max length of sequence
def max_length(sequences):
    '''
    input:
        sequences: a 2D list of integer sequences
    output:
        max_length: the max length of the sequences
    '''
```

```python
    max_length = 0
    for i, seq in enumerate(sequences):
        length = len(seq)
        if max_length < length:
            max_length = length
    return max_length

# Calculated the statistics of Word2Vec
emb_mean = word2vec.vectors.mean()
emb_std = word2vec.vectors.std()

# Map the Word2Vec into a matrix for embedding weights
def pretrained_embedding_matrix(word_to_vec_map,
                                word_to_index,
                                emb_mean, emb_std):

    np.random.seed(2021)

    # adding 1 to fit Keras embedding (requirement)
    vocab_size = len(word_to_index) + 1
    # define dimensionality of your pre-trained word vectors
    emb_dim = word_to_vec_map.word_vec('handsome').shape[0]

    # initialize the matrix with generic normal distribution
    embed_matrix = np.random.normal(emb_mean,
                                    emb_std,
                                    (vocab_size, emb_dim))

    # Set each row "idx" of the embedding matrix to be
    # the word vector representation of the idx'th
    # word of the vocabulary
    for word, idx in word_to_index.items():
        if word in word_to_vec_map:

            embed_matrix[idx] = word_to_vec_map.get_vector(word)

    return embed_matrix

#===============================Step 3============================#
# Model Definitions

from tensorflow.keras import regularizers
from tensorflow.keras.constraints import MaxNorm

def define_model(input_dim = None,
                 output_dim=300,
                 max_length = None ):

    model = tf.keras.models.Sequential([
        tf.keras.layers.Embedding(input_dim=input_dim,
                                  mask_zero= True,
                                  output_dim=output_dim,
                                  input_length=max_length,
                                  input_shape=(max_length, )),
```

```
        # Change it to tf.keras.layers.LSTM(64) for LSTM version
        # inside the Bidirectional layer
        tf.keras.layers.Bidirectional((tf.keras.layers.GRU(64))),
        tf.keras.layers.Dropout(0.5),
        # Propagate X through a Dense layer with 1 unit
        tf.keras.layers.Dense(units=1, activation='sigmoid')
    ])

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

    return model

from tensorflow.keras import regularizers
from tensorflow.keras.constraints import MaxNorm

def define_model_2(input_dim = None,
                   output_dim=300,
                   max_length = None,
                   emb_matrix=None):

    model = tf.keras.models.Sequential([
        tf.keras.layers.Embedding(input_dim=input_dim,
                                  mask_zero= True,
                                  output_dim=output_dim,
                                  input_length=max_length,
                                  input_shape=(max_length, ),
                                  # Assign the embedding weight
                                  # with word2vec embedding marix
                                  weights = [emb_matrix],
                                  # Set the weight to be not
                                  # trainable (static)
                                  trainable = False),

        # Change it to tf.keras.layers.LSTM(64) for LSTM version
        # inside the Bidirectional layer
        tf.keras.layers.Bidirectional((tf.keras.layers.GRU(64))),
        tf.keras.layers.Dropout(0.5),
        # Propagate X through a Dense layer with 1 unit
        tf.keras.layers.Dense(units=1, activation='sigmoid')
    ])

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

    return model

def define_model_3(input_dim = None,
                   output_dim=300,
                   max_length = None,
                   emb_matrix=None):

    model = tf.keras.models.Sequential([
```

```
        tf.keras.layers.Embedding(input_dim=input_dim,
                                  mask_zero= True,
                                  output_dim=output_dim,
                                  input_length=max_length,
                                  input_shape=(max_length, ),
                                  # Assign the embedding weight
                                  # with word2vec embedding marix
                                  weights = [emb_matrix],
                                  # Set the weight to be not
                                  # trainable (static)
                                  trainable = True),

        # Change it to tf.keras.layers.LSTM(64) for LSTM version
        # inside the Bidirectional layer
        tf.keras.layers.Bidirectional((tf.keras.layers.GRU(64))),
        tf.keras.layers.Dropout(0.5),
        # Propagate X through a Dense layer with 1 unit
        tf.keras.layers.Dense(units=1, activation='sigmoid')
    ])

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

    return model

# Define the early stopping callbacks
callbacks = tf.keras.callbacks.EarlyStopping(
            monitor='val_accuracy',
            min_delta=0,
            patience=10, verbose=2,
            mode='auto',
            restore_best_weights=True)

#==============================Step 3============================#
# Begin the training process with random/static/dynamic Word2Vec

# Parameter Initialization
trunc_type='post'
padding_type='post'
oov_tok = "<UNK>"
activations = ['ReLU', 'tanh']
filters = 100
kernel_sizes = [1, 2, 3, 4, 5, 6]
emb_mean = emb_mean
emb_std = emb_std

columns = ['Activation', 'Filters', 'acc1', 'acc2', 'acc3',
           'acc4', 'acc5', 'acc6', 'acc7', 'acc8', 'acc9',
           'acc10', 'AVG']

record_1 = pd.DataFrame(columns = columns)
record_2 = pd.DataFrame(columns = columns)
record_3 = pd.DataFrame(columns = columns)
```

```
# prepare cross validation with 10 splits and shuffle = True
kfold = KFold(10, True)

# Separate the sentences and the labels
sentences, labels = list(corpus.sentence), list(corpus.label)

for activation in activations:
    for kernel_size in kernel_sizes:
        # kfold.split() will return set indices for each split
        acc_list_1 = []
        acc_list_2 = []
        acc_list_3 = []

        for train, test in kfold.split(sentences):

            train_x, test_x = [], []
            train_y, test_y = [], []

            for i in train:
                train_x.append(sentences[i])
                train_y.append(labels[i])

            for i in test:
                test_x.append(sentences[i])
                test_y.append(labels[i])

            # Turn the labels into a numpy array
            train_y = np.array(train_y)
            test_y = np.array(test_y)

            # encode data using
            # Cleaning and Tokenization
            tokenizer = Tokenizer(oov_token=oov_tok)
            tokenizer.fit_on_texts(train_x)

            # Turn the text into sequence
            training_sequences = tokenizer.texts_to_sequences(
                                     train_x)
            test_sequences = tokenizer.texts_to_sequences(
                                 test_x)

            max_len = max_length(training_sequences)

            # Pad the sequence to have the same size
            Xtrain = pad_sequences(training_sequences,
                                      maxlen=max_len,
                                      padding=padding_type,
                                      truncating=trunc_type)
            Xtest = pad_sequences(test_sequences,
                                    maxlen=max_len,
                                    padding=padding_type,
                                    truncating=trunc_type)

            word_index = tokenizer.word_index
            vocab_size = len(word_index)+1
```

```
    emb_matrix = pretrained_embedding_matrix(word2vec,
                                             word_index,
                                             emb_mean,
                                             emb_std)

    # Define the models to train
    model_1 = define_model(filters,
                           kernel_size,
                           activation,
                           input_dim=vocab_size,
                           max_length=max_len)

    model_2 = define_model_2(filters,
                             kernel_size,
                             activation,
                             input_dim=vocab_size,
                             max_length=max_len,
                             emb_matrix=emb_matrix)

    model_3 = define_model_3(filters,
                             kernel_size,
                             activation,
                             input_dim=vocab_size,
                             max_length=max_len,
                             emb_matrix=emb_matrix)


    # Train the models
    model_1.fit(Xtrain, train_y, batch_size=50,
                epochs=100, verbose=1,
                callbacks=[callbacks],
                validation_data=(Xtest, test_y))

    model_2.fit(Xtrain, train_y, batch_size=50,
                epochs=100, verbose=1,
                callbacks=[callbacks],
                validation_data=(Xtest, test_y))

    model_3.fit(Xtrain, train_y, batch_size=50,
                epochs=100, verbose=1,
              callbacks=[callbacks],
                validation_data=(Xtest, test_y))

    # evaluate the model
    loss_1, acc_1 = model_1.evaluate(Xtest, test_y)
    loss_2, acc_2 = model_2.evaluate(Xtest, test_y)
    loss_3, acc_3 = model_3.evaluate(Xtest, test_y)

    acc_list_1.append(acc*100)
    acc_list_2.append(acc*100)
    acc_list_3.append(acc*100)

mean_acc_1 = np.array(acc_list_1).mean()
```

```python
        mean_acc_2 = np.array(acc_list_2).mean()
        mean_acc_3 = np.array(acc_list_3).mean()

        parameters = [activation, kernel_size]
        entries_1 = parameters + acc_list + [mean_acc_1]
        entries_2 = parameters + acc_list + [mean_acc_2]
        entries_3 = parameters + acc_list + [mean_acc_3]

        temp = pd.DataFrame([entries_1], columns=columns)
        record_1 = record_1.append(temp, ignore_index=True)
        temp = pd.DataFrame([entries_2], columns=columns)
        record_2 = record_2.append(temp, ignore_index=True)
        temp = pd.DataFrame([entries_3], columns=columns)
        record_3 = record_3.append(temp, ignore_index=True)

        print(record_1)
        print(record_2)
        print(record_3)

#==============================Step 5=============================#
# Save the dataframe into excel file
record_1.to_excel('WE_GRU/LSTM_1.xlsx', sheet_name='random')
record_2.to_excel('WE_GRU/LSTM_2.xlsx', sheet_name='static')
record_3.to_excel('WE_GRU/LSTM_3.xlsx', sheet_name='dynamic')
#=================================================================#
```

Appendix E

Python Program for Stacked BiGRU/BiLSTM Models

```
"""
Title             : Python Program for Stacked BiGRU/BiLSTM Models
Dataset           : CR (as an example)
Feature Extration: Random, Static, Dynamic Word2Vec
Author            : Diardano Raihan
"""

##===========================Import Libraries=======================#
import re
import numpy as np
import pandas as pd
import tensorflow as tf
from string import punctuation
from collections import Counter
import matplotlib.pyplot as plt
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from gensim.models import KeyedVectors
from sklearn.model_selection import KFold
from tensorflow.keras import regularizers
from tensorflow.keras.constraints import MaxNorm
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

#==============================Step 1=============================#
# Load the dataset
corpus = pd.read_pickle('../../0_data/CR/CR.pkl')
# Load the Word2Vec
word2vec = KeyedVectors.load_word2vec_format(
            '../GoogleNews-vectors-negative300.bin',
            binary=True)

#==============================Step 2=============================#
# Defined all the functions needed as Text Preprocessing steps

# Define a function to compute the max length of sequence
def max_length(sequences):
    '''
    input:
        sequences: a 2D list of integer sequences
    output:
        max_length: the max length of the sequences
    '''
```

```
    max_length = 0
    for i, seq in enumerate(sequences):
        length = len(seq)
        if max_length < length:
            max_length = length
    return max_length

# Calculated the statistics of Word2Vec
emb_mean = word2vec.vectors.mean()
emb_std = word2vec.vectors.std()

# Map the Word2Vec into a matrix for embedding weights
def pretrained_embedding_matrix(word_to_vec_map,
                                word_to_index,
                                emb_mean, emb_std):

    np.random.seed(2021)

    # adding 1 to fit Keras embedding (requirement)
    vocab_size = len(word_to_index) + 1
    # define dimensionality of your pre-trained word vectors
    emb_dim = word_to_vec_map.word_vec('handsome').shape[0]

    # initialize the matrix with generic normal distribution
    embed_matrix = np.random.normal(emb_mean,
                                    emb_std,
                                    (vocab_size, emb_dim))

    # Set each row "idx" of the embedding matrix to be
    # the word vector representation of the idx'th
    # word of the vocabulary
    for word, idx in word_to_index.items():
        if word in word_to_vec_map:

            embed_matrix[idx] = word_to_vec_map.get_vector(word)

    return embed_matrix

#===============================Step 3============================#
# Model Definitions

from tensorflow.keras import regularizers
from tensorflow.keras.constraints import MaxNorm

def define_model(input_dim = None,
                 output_dim=300,
                 max_length = None ):

    model = tf.keras.models.Sequential([
        tf.keras.layers.Embedding(input_dim=input_dim,
                                  mask_zero= True,
                                  output_dim=output_dim,
                                  input_length=max_length,
                                  input_shape=(max_length, )),
```

```
        # Change it to tf.keras.layers.LSTM(64) for LSTM version
        # inside the Bidirectional layer
        tf.keras.layers.Bidirectional(tf.keras.layers.GRU(
                                              64,
                                              return_sequences=True)),
        tf.keras.layers.Bidirectional(tf.keras.layers.GRU(
                                              64,
                                              return_sequences=False)),
        tf.keras.layers.Dropout(0.5),
        # Propagate X through a Dense layer with 1 unit
        tf.keras.layers.Dense(units=1, activation='sigmoid')
    ])

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

    return model

from tensorflow.keras import regularizers
from tensorflow.keras.constraints import MaxNorm

def define_model_2(input_dim = None,
                   output_dim=300,
                   max_length = None,
                   emb_matrix=None):

    model = tf.keras.models.Sequential([
        tf.keras.layers.Embedding(input_dim=input_dim,
                                  mask_zero= True,
                                  output_dim=output_dim,
                                  input_length=max_length,
                                  input_shape=(max_length, ),
                                  # Assign the embedding weight
                                  # with word2vec embedding marix
                                  weights = [emb_matrix],
                                  # Set the weight to be not
                                  # trainable (static)
                                  trainable = False),

        # Change it to tf.keras.layers.LSTM(64) for LSTM version
        # inside the Bidirectional layer
        tf.keras.layers.Bidirectional(tf.keras.layers.GRU(
                                              64,
                                              return_sequences=True)),
        tf.keras.layers.Bidirectional(tf.keras.layers.GRU(
                                              64,
                                              return_sequences=False)),
        tf.keras.layers.Dropout(0.5),
        # Propagate X through a Dense layer with 1 unit
        tf.keras.layers.Dense(units=1, activation='sigmoid')
    ])

    model.compile(loss = 'binary_crossentropy',
                  optimizer = 'adam',
```

```
                 metrics = ['accuracy'])

    return model

def define_model_3(input_dim = None,
                   output_dim=300,
                   max_length = None,
                   emb_matrix=None):

    model = tf.keras.models.Sequential([
        tf.keras.layers.Embedding(input_dim=input_dim,
                                  mask_zero= True,
                                  output_dim=output_dim,
                                  input_length=max_length,
                                  input_shape=(max_length, ),
                                  # Assign the embedding weight
                                  # with word2vec embedding marix
                                  weights = [emb_matrix],
                                  # Set the weight to be not
                                  # trainable (static)
                                  trainable = True),

        # Change it to tf.keras.layers.LSTM(64) for LSTM version
        # inside the Bidirectional layer
        tf.keras.layers.Bidirectional(tf.keras.layers.GRU(
                                      64,
                                      return_sequences=True)),
        tf.keras.layers.Bidirectional(tf.keras.layers.GRU(
                                      64,
                                      return_sequences=False)),
        tf.keras.layers.Dropout(0.5),
        # Propagate X through a Dense layer with 1 unit
        tf.keras.layers.Dense(units=1, activation='sigmoid')
    ])

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

    return model

# Define the early stopping callbacks
callbacks = tf.keras.callbacks.EarlyStopping(
            monitor='val_accuracy',
            min_delta=0,
            patience=10, verbose=2,
            mode='auto',
            restore_best_weights=True)

#===============================Step 3=============================#
# Begin the training process with random/static/dynamic Word2Vec

# Parameter Initialization
trunc_type='post'
padding_type='post'
```

```python
oov_tok = "<UNK>"
activations = ['ReLU', 'tanh']
filters = 100
kernel_sizes = [1, 2, 3, 4, 5, 6]
emb_mean = emb_mean
emb_std = emb_std

columns = ['Activation', 'Filters', 'acc1', 'acc2', 'acc3',
           'acc4', 'acc5', 'acc6', 'acc7', 'acc8', 'acc9',
           'acc10', 'AVG']

record_1 = pd.DataFrame(columns = columns)
record_2 = pd.DataFrame(columns = columns)
record_3 = pd.DataFrame(columns = columns)

# prepare cross validation with 10 splits and shuffle = True
kfold = KFold(10, True)

# Separate the sentences and the labels
sentences, labels = list(corpus.sentence), list(corpus.label)

for activation in activations:
    for kernel_size in kernel_sizes:
        # kfold.split() will return set indices for each split
        acc_list_1 = []
        acc_list_2 = []
        acc_list_3 = []

        for train, test in kfold.split(sentences):

            train_x, test_x = [], []
            train_y, test_y = [], []

            for i in train:
                train_x.append(sentences[i])
                train_y.append(labels[i])

            for i in test:
                test_x.append(sentences[i])
                test_y.append(labels[i])

            # Turn the labels into a numpy array
            train_y = np.array(train_y)
            test_y = np.array(test_y)

            # encode data using
            # Cleaning and Tokenization
            tokenizer = Tokenizer(oov_token=oov_tok)
            tokenizer.fit_on_texts(train_x)

            # Turn the text into sequence
            training_sequences = tokenizer.texts_to_sequences(
                                     train_x)
            test_sequences = tokenizer.texts_to_sequences(
                                 test_x)
```

```
max_len = max_length(training_sequences)

# Pad the sequence to have the same size
Xtrain = pad_sequences(training_sequences,
                       maxlen=max_len,
                       padding=padding_type,
                       truncating=trunc_type)
Xtest = pad_sequences(test_sequences,
                      maxlen=max_len,
                      padding=padding_type,
                      truncating=trunc_type)

word_index = tokenizer.word_index
vocab_size = len(word_index)+1


emb_matrix = pretrained_embedding_matrix(word2vec,
                                         word_index,
                                         emb_mean,
                                         emb_std)

# Define the models to train
model_1 = define_model(filters,
                       kernel_size,
                       activation,
                       input_dim=vocab_size,
                       max_length=max_len)

model_2 = define_model_2(filters,
                         kernel_size,
                         activation,
                         input_dim=vocab_size,
                         max_length=max_len,
                         emb_matrix=emb_matrix)

model_3 = define_model_3(filters,
                         kernel_size,
                         activation,
                         input_dim=vocab_size,
                         max_length=max_len,
                         emb_matrix=emb_matrix)


# Train the models
model_1.fit(Xtrain, train_y, batch_size=50,
            epochs=100, verbose=1,
            callbacks=[callbacks],
            validation_data=(Xtest, test_y))

model_2.fit(Xtrain, train_y, batch_size=50,
            epochs=100, verbose=1,
            callbacks=[callbacks],
            validation_data=(Xtest, test_y))
```

```
            model_3.fit(Xtrain, train_y, batch_size=50,
                        epochs=100, verbose=1,
                      callbacks=[callbacks],
                        validation_data=(Xtest, test_y))

            # evaluate the model
            loss_1, acc_1 = model_1.evaluate(Xtest, test_y)
            loss_2, acc_2 = model_2.evaluate(Xtest, test_y)
            loss_3, acc_3 = model_3.evaluate(Xtest, test_y)

            acc_list_1.append(acc*100)
            acc_list_2.append(acc*100)
            acc_list_3.append(acc*100)

        mean_acc_1 = np.array(acc_list_1).mean()
        mean_acc_2 = np.array(acc_list_2).mean()
        mean_acc_3 = np.array(acc_list_3).mean()

        parameters = [activation, kernel_size]
        entries_1 = parameters + acc_list + [mean_acc_1]
        entries_2 = parameters + acc_list + [mean_acc_2]
        entries_3 = parameters + acc_list + [mean_acc_3]

        temp = pd.DataFrame([entries_1], columns=columns)
        record_1 = record_1.append(temp, ignore_index=True)
        temp = pd.DataFrame([entries_2], columns=columns)
        record_2 = record_2.append(temp, ignore_index=True)
        temp = pd.DataFrame([entries_3], columns=columns)
        record_3 = record_3.append(temp, ignore_index=True)

        print(record_1)
        print(record_2)
        print(record_3)

#==============================Step 5============================#
# Save the dataframe into excel file
record_1.to_excel('WE_StackedBiGRU/BiLSTM_1.xlsx',
                  sheet_name='random')
record_2.to_excel('WE_StackedBiGRU/BiLSTM_2.xlsx',
                  sheet_name='static')
record_3.to_excel('WE_StackedBiGRU/BiLSTM_3.xlsx',
                  sheet_name='dynamic')
#================================================================#
```

Appendix F

Python Program for Ensemble Learning-based Models

```
"""
Title             : Python Program for Ensemble-based Models
Dataset           : CR (as an example)
Feature Extration: Random, Static, Dynamic Word2Vec
Author            : Diardano Raihan
"""

##=========================Import Libraries=======================#
import re
import numpy as np
import pandas as pd
import tensorflow as tf
from string import punctuation
from collections import Counter
import matplotlib.pyplot as plt
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from gensim.models import KeyedVectors
from sklearn.model_selection import KFold
from tensorflow.keras import regularizers
from tensorflow.keras.constraints import MaxNorm
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

from tensorflow.keras.layers import Input, Embedding, Conv1D
from tensorflow.keras.layers import Dropout, MaxPool1D, Flatten
from tensorflow.keras.layers import Dense, Bidirectional, GRU
from tensorflow.keras.models import Model
from tensorflow.keras.layers import concatenate

#==============================Step 1============================#
# Load the dataset
corpus = pd.read_pickle('../../0_data/CR/CR.pkl')
# Load the Word2Vec
word2vec = KeyedVectors.load_word2vec_format(
            '../GoogleNews-vectors-negative300.bin',
            binary=True)

#==============================Step 2============================#
# Defined all the functions needed as Text Preprocessing steps

# Define a function to compute the max length of sequence
def max_length(sequences):
```

```
    '''
    input:
        sequences: a 2D list of integer sequences
    output:
        max_length: the max length of the sequences
    '''
    max_length = 0
    for i, seq in enumerate(sequences):
        length = len(seq)
        if max_length < length:
            max_length = length
    return max_length

# Calculated the statistics of Word2Vec
emb_mean = word2vec.vectors.mean()
emb_std = word2vec.vectors.std()

# Map the Word2Vec into a matrix for embedding weights
def pretrained_embedding_matrix(word_to_vec_map,
                                word_to_index,
                                emb_mean, emb_std):

    np.random.seed(2021)

    # adding 1 to fit Keras embedding (requirement)
    vocab_size = len(word_to_index) + 1
    # define dimensionality of your pre-trained word vectors
    emb_dim = word_to_vec_map.word_vec('handsome').shape[0]

    # initialize the matrix with generic normal distribution
    embed_matrix = np.random.normal(emb_mean,
                                    emb_std,
                                    (vocab_size, emb_dim))

    # Set each row "idx" of the embedding matrix to be
    # the word vector representation of the idx'th
    # word of the vocabulary
    for word, idx in word_to_index.items():
        if word in word_to_vec_map:

            embed_matrix[idx] = word_to_vec_map.get_vector(word)

    return embed_matrix

#===============================Step 3=============================#
# Model Definitions

def define_model(filters = 100, kernel_size = 3,
                 activation='ReLU', input_dim = None,
                 output_dim=300, max_length = None ):

    # Channel 1
    input1 = Input(shape=(max_length,))
    embeddding1 = Embedding(input_dim=input_dim,
                            output_dim=output_dim,
```

```
                             input_length=max_length)(input1)
    conv1 = Conv1D(filters=filters,
                   kernel_size=kernel_size,
                   activation='ReLU',
                   kernel_constraint= MaxNorm(max_value=3,
                                              axis=[0,1])
                  )(embeddding1)
    pool1 = MaxPool1D(pool_size=2, strides=2)(conv1)
    flat1 = Flatten()(pool1)
    drop1 = Dropout(0.5)(flat1)
    dense1 = Dense(10, activation='ReLU')(drop1)
    drop1 = Dropout(0.5)(dense1)
    out1 = Dense(1, activation='sigmoid')(drop1)

    # Channel 2
    input2 = Input(shape=(max_length,))
    embeddding2 = Embedding(input_dim=input_dim,
                            output_dim=output_dim,
                            input_length=max_length,
                            mask_zero=True)(input2)

    gru2 = Bidirectional(GRU(64))(embeddding2)
    drop2 = Dropout(0.5)(gru2)
    out2 = Dense(1, activation='sigmoid')(drop2)

    # Merge
    merged = concatenate([out1, out2])

    # Interpretation
    outputs = Dense(1, activation='sigmoid')(merged)
    model = Model(inputs=[input1, input2], outputs=outputs)

    # Compile
    model.compile( loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])

    return model

def define_model_2(filters = 100, kernel_size = 3,
                   activation='ReLU',
                   input_dim = None, output_dim=300,
                   max_length = None, emb_matrix = None):

    # Channel 1
    input1 = Input(shape=(max_length,))
    embeddding1 = Embedding(input_dim=input_dim,
                            output_dim=output_dim,
                            input_length=max_length,
                            input_shape=(max_length, ),
                            # Assign the embedding weight
                            # with word2vec embedding marix
                            weights = [emb_matrix],
                            # Set the weight to be not trainable
                            # (static)
```

```
                            trainable = False)(input1)
    conv1 = Conv1D(filters=filters,
                   kernel_size=kernel_size,
                   activation='ReLU',
                   kernel_constraint= MaxNorm(max_value=3,
                                              axis=[0,1])
                  )(embeddding1)
    pool1 = MaxPool1D(pool_size=2, strides=2)(conv1)
    flat1 = Flatten()(pool1)
    drop1 = Dropout(0.5)(flat1)
    dense1 = Dense(10, activation='ReLU')(drop1)
    drop1 = Dropout(0.5)(dense1)
    out1 = Dense(1, activation='sigmoid')(drop1)

    # Channel 2
    input2 = Input(shape=(max_length,))
    embeddding2 = Embedding(input_dim=input_dim,
                            output_dim=output_dim,
                            input_length=max_length,
                            input_shape=(max_length, ),
                            # Assign the embedding weight
                            # with word2vec embedding marix
                            weights = [emb_matrix],
                            # Set the weight to be not trainable
                            # (static)
                            trainable = False,
                            mask_zero=True)(input2)

    gru2 = Bidirectional(GRU(64))(embeddding2)
    drop2 = Dropout(0.5)(gru2)
    out2 = Dense(1, activation='sigmoid')(drop2)

    # Merge
    merged = concatenate([out1, out2])

    # Interpretation
    outputs = Dense(1, activation='sigmoid')(merged)
    model = Model(inputs=[input1, input2], outputs=outputs)

    # Compile
    model.compile( loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])

    return model

def define_model_3(filters = 100, kernel_size = 3,
                   activation='ReLU',
                   input_dim = None, output_dim=300,
                   max_length = None, emb_matrix = None):

    # Channel 1
    input1 = Input(shape=(max_length,))
    embeddding1 = Embedding(input_dim=input_dim,
                            output_dim=output_dim,
```

```
                                input_length=max_length,
                                input_shape=(max_length, ),
                                # Assign the embedding weight
                                # with word2vec embedding marix
                                weights = [emb_matrix],
                                # Set the weight to be not trainable
                                # (static)
                                trainable = True)(input1)
    conv1 = Conv1D(filters=filters,
                   kernel_size=kernel_size,
                   activation='ReLU',
                   kernel_constraint= MaxNorm(max_value=3,
                                              axis=[0,1])
                  )(embeddding1)
    pool1 = MaxPool1D(pool_size=2, strides=2)(conv1)
    flat1 = Flatten()(pool1)
    drop1 = Dropout(0.5)(flat1)
    dense1 = Dense(10, activation='ReLU')(drop1)
    drop1 = Dropout(0.5)(dense1)
    out1 = Dense(1, activation='sigmoid')(drop1)

    # Channel 2
    input2 = Input(shape=(max_length,))
    embeddding2 = Embedding(input_dim=input_dim,
                                output_dim=output_dim,
                                input_length=max_length,
                                input_shape=(max_length, ),
                                # Assign the embedding weight
                                # with word2vec embedding marix
                                weights = [emb_matrix],
                                # Set the weight to be not trainable
                                # (static)
                                trainable = True,
                                mask_zero=True)(input2)

    gru2 = Bidirectional(GRU(64))(embeddding2)
    drop2 = Dropout(0.5)(gru2)
    out2 = Dense(1, activation='sigmoid')(drop2)

    # Merge
    merged = concatenate([out1, out2])

    # Interpretation
    outputs = Dense(1, activation='sigmoid')(merged)
    model = Model(inputs=[input1, input2], outputs=outputs)

    # Compile
    model.compile( loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])

    return model

# Define the early stopping callbacks
callbacks = tf.keras.callbacks.EarlyStopping(
```

```
            monitor='val_accuracy',
            min_delta=0,
            patience=10, verbose=2,
            mode='auto',
            restore_best_weights=True)

#================================Step 3=============================#
# Begin the training process with random/static/dynamic Word2Vec

# Parameter Initialization
trunc_type='post'
padding_type='post'
oov_tok = "<UNK>"
activations = ['ReLU', 'tanh']
filters = 100
kernel_sizes = [1, 2, 3, 4, 5, 6]
emb_mean = emb_mean
emb_std = emb_std

columns = ['Activation', 'Filters', 'acc1', 'acc2', 'acc3',
           'acc4', 'acc5', 'acc6', 'acc7', 'acc8', 'acc9',
           'acc10', 'AVG']

record_1 = pd.DataFrame(columns = columns)
record_2 = pd.DataFrame(columns = columns)
record_3 = pd.DataFrame(columns = columns)

# prepare cross validation with 10 splits and shuffle = True
kfold = KFold(10, True)

# Separate the sentences and the labels
sentences, labels = list(corpus.sentence), list(corpus.label)

for activation in activations:
    for kernel_size in kernel_sizes:
        # kfold.split() will return set indices for each split
        acc_list_1 = []
        acc_list_2 = []
        acc_list_3 = []

        for train, test in kfold.split(sentences):

            train_x, test_x = [], []
            train_y, test_y = [], []

            for i in train:
                train_x.append(sentences[i])
                train_y.append(labels[i])

            for i in test:
                test_x.append(sentences[i])
                test_y.append(labels[i])

            # Turn the labels into a numpy array
            train_y = np.array(train_y)
```

```python
test_y = np.array(test_y)

# encode data using
# Cleaning and Tokenization
tokenizer = Tokenizer(oov_token=oov_tok)
tokenizer.fit_on_texts(train_x)

# Turn the text into sequence
training_sequences = tokenizer.texts_to_sequences(
                     train_x)
test_sequences = tokenizer.texts_to_sequences(
                 test_x)

max_len = max_length(training_sequences)

# Pad the sequence to have the same size
Xtrain = pad_sequences(training_sequences,
                       maxlen=max_len,
                       padding=padding_type,
                       truncating=trunc_type)
Xtest = pad_sequences(test_sequences,
                      maxlen=max_len,
                      padding=padding_type,
                      truncating=trunc_type)

word_index = tokenizer.word_index
vocab_size = len(word_index)+1


emb_matrix = pretrained_embedding_matrix(word2vec,
                                         word_index,
                                         emb_mean,
                                         emb_std)

# Define the models to train
model_1 = define_model(filters,
                       kernel_size,
                       activation,
                       input_dim=vocab_size,
                       max_length=max_len)

model_2 = define_model_2(filters,
                         kernel_size,
                         activation,
                         input_dim=vocab_size,
                         max_length=max_len,
                         emb_matrix=emb_matrix)

model_3 = define_model_3(filters,
                         kernel_size,
                         activation,
                         input_dim=vocab_size,
                         max_length=max_len,
                         emb_matrix=emb_matrix)
```

```python
        # Train the models
        model_1.fit(x=[Xtrain, Xtrain], train_y, batch_size=50,
                    epochs=100, verbose=1,
                    callbacks=[callbacks],
                    validation_data=([Xtest, Xtest], test_y))

        model_2.fit(x=[Xtrain, Xtrain], train_y, batch_size=50,
                    epochs=100, verbose=1,
                    callbacks=[callbacks],
                    validation_data=([Xtest, Xtest], test_y))

        model_3.fit(x=[Xtrain, Xtrain], train_y, batch_size=50,
                    epochs=100, verbose=1,
                  callbacks=[callbacks],
                    validation_data=([Xtest, Xtest], test_y))

        # evaluate the model
        loss_1, acc_1 = model_1.evaluate([Xtest, Xtest], test_y)
        loss_2, acc_2 = model_2.evaluate([Xtest, Xtest], test_y)
        loss_3, acc_3 = model_3.evaluate([Xtest, Xtest], test_y)

        acc_list_1.append(acc*100)
        acc_list_2.append(acc*100)
        acc_list_3.append(acc*100)

    mean_acc_1 = np.array(acc_list_1).mean()
    mean_acc_2 = np.array(acc_list_2).mean()
    mean_acc_3 = np.array(acc_list_3).mean()

    parameters = [activation, kernel_size]
    entries_1 = parameters + acc_list + [mean_acc_1]
    entries_2 = parameters + acc_list + [mean_acc_2]
    entries_3 = parameters + acc_list + [mean_acc_3]

    temp = pd.DataFrame([entries_1], columns=columns)
    record_1 = record_1.append(temp, ignore_index=True)
    temp = pd.DataFrame([entries_2], columns=columns)
    record_2 = record_2.append(temp, ignore_index=True)
    temp = pd.DataFrame([entries_3], columns=columns)
    record_3 = record_3.append(temp, ignore_index=True)

    print(record_1)
    print(record_2)
    print(record_3)

#==============================Step 5============================#
# Save the dataframe into excel file
record_1.to_excel('WE_Ensemble_1.xlsx', sheet_name='random')
record_2.to_excel('WE_Ensemble_2.xlsx', sheet_name='static')
record_3.to_excel('WE_Ensemble_3.xlsx', sheet_name='dynamic')
#================================================================#
```

Appendix G

Matlab Program for EDRVFL Models

Note: *This code is maintained by Nanyang Technological University*

- File: **MAIN.m**

```
clc;
clear;

% Load datasets (The string inside the load function is the path to
the
% .mat file)
dataset = load('BoW_datasets/CR/binary/binary_CR_10.mat');
% test_set = load('MatDataset/abalone/abalone_Test.mat');

% Collect training/testing datas et
trainX = dataset.Xtrain;
trainY = dataset.ytrain.';
testX = dataset.Xtest;
testY = dataset.ytest.';

% Encode the label
[trainYT,classes] = OneVAllEncode(trainY);
testYT = OneVAllEncode(testY,classes);

% Parameters
ModelParameters.L = 10;                  % Number of layers
ModelParameters.N = 100;                 % Number of neurons
ModelParameters.C = 0.1;                 % Regularisation Parameter
ModelParameters.scale = 1;               % Scaling parameter
ModelParameters.Activation = "SELU";     % Activation function
% ReLU, sigmoid, SELU, radbas, sine


%% Tuning
% Parameters to tune (You can also experiment with different values)
N_range = 3:20:183;
C_range = 2.^(-5:1:3);

best_acc = 0;    % Initialisation

% Requried for consistency
s = RandStream('mcg16807','Seed',0);
RandStream.setGlobalStream(s);
```

```
cv_part = cvpartition(trainY,'KFold',4);     % Create indices for
training/validation subsets

% Test every configuration

for p1 = 1:numel(N_range)
    for p2 = 1:numel(C_range)
        TestModelParameters = ModelParameters;
        TestModelParameters.N = N_range(p1);
        TestModelParameters.C = C_range(p2);

        val_acc = zeros(4,1);    % Initialisation
        for k = 1:4
            % Collect training/validation sets
            val_trainX = trainX(cv_part.training(k),:);
            val_trainY = trainYT(cv_part.training(k),:);
            val_testX = trainX(cv_part.test(k),:);
            val_testY = trainYT(cv_part.test(k),:);

            % Data Normalisation
%             mean_X = mean(val_trainX,1);
%             std_X = std(val_trainX);
%             std_X(std_X==0) = 1e-4;                  % For
numerical stability
%             val_trainX = bsxfun(@rdivide,val_trainX-
repmat(mean_X,size(val_trainX,1),1),std_X);
%             val_testX = bsxfun(@rdivide,val_testX-
repmat(mean_X,size(val_testX,1),1),std_X);

            % Training and Testing
            [~,~,val_acc(k),~,~] =
MRVFL(val_trainX,val_trainY,val_testX,val_testY,TestModelParameters)
;

        end

        % Average the validation accuracy
        ValAcc = mean(val_acc);

        % Check if current configuration is the best
        if ValAcc > best_acc
            best_acc = ValAcc;
            best_N = N_range(p1);
            best_C = C_range(p2);
        end
    end
end

% Use the best settings
ModelParameters.N = best_N;
ModelParameters.C = best_C;


%% Evaluation
% Data Normalisation
```

```
% mean_X = mean(trainX,1);
% std_X = std(trainX);
% std_X(std_X==0) = 1;                    % For numerical stability
% trainX = bsxfun(@rdivide,trainX-
repmat(mean_X,size(trainX,1),1),std_X);
% testX = bsxfun(@rdivide,testX-
repmat(mean_X,size(testX,1),1),std_X);

[Model,TrainAcc,TestAcc,TrainingTime,TestingTime] =
MRVFL(trainX,trainYT,testX,testYT,ModelParameters);
```

- File: **MRVFL.m**

```
function [Model,TrainAcc,TestAcc,TrainingTime,TestingTime]  =
MRVFL(trainX,trainY,testX,testY,option)

% Requried for consistency
s = RandStream('mcg16807','Seed',0);
RandStream.setGlobalStream(s);

% Train RVFL
[Model,TrainAcc,TrainingTime] = MRVFLtrain(trainX,trainY,option);

% Using trained model, predict the testing data
[TestAcc,TestingTime] = MRVFLpredict(testX,testY,Model);

end
%EOF
```

- File: **MRVFLtrain.m**

```
function [model,TrainingAccuracy,Training_time] =
MRVFLtrain(trainX,trainY,option)

% Parameters
[Nsample,Nfea] = size(trainX);
N = option.N;
L = option.L;
C = option.C;
s = option.scale;  %scaling factor
act_fun = option.Activation;

% Initialization
A = cell(L,1); %for L hidden layers
beta = cell(L,1);
weights = cell(L,1);
biases = cell(L,1);

A_input = trainX;

% Loop for each layer
tic
for i = 1:L
```

```
% Hidden Layer randomization
if i==1
    w = s*(2*rand(Nfea,N)-1);
else
    w = s*(2*rand(Nfea+N+1,N)-1);
end
b = s*rand(1,N);

% Store for future use
weights{i} = w;
biases{i} = b;

% Application
A1 = A_input * w;

% Standard Normalization for every input feature
mu{i} = mean(A1,1);
sigma{i} = std(A1);
A1 = bsxfun(@rdivide,A1-repmat(mu{i},size(A1,1),1),sigma{i});

% Apply random bias
A1 = A1+repmat(b,Nsample,1);

% Activation Function
switch lower(act_fun)
    case 'ReLU' % Range: [0,inf]
        A1 = ReLU(A1);

    case {'sig','sigmoid'} % Range: [0,1]
        A1 = sigmoid(A1);

    case {'sin','sine'} % Range: [-1,1]
        A1 = sin(A1);

    case 'hardlim' % Range: [0,1]
        A1 = double(hardlim(A1));

    case 'tribas' % Range: [0,1]
        A1 = tribas(A1);

    case 'radbas' % Range: [0,1]
        A1 = radbas(A1);

    case 'sign' % Range: [-1,1]
        A1 = sign(A1);

    case 'SELU' % Range: [-1,inf]
        A1 = SELU(A1);

    otherwise
        error('Activation function not recognized.');

end
```

```
    % Output to classifier layer
    A1_temp1 = [A_input,A1,ones(Nsample,1)];        % Direct links +
bias to the output layer
    beta1  = l2_weights(A1_temp1,trainY,C,Nsample);

    % Store for future use
    A{i} = A1_temp1;
    beta{i} = beta1;

    % Output to next hidden layer
    A1_temp3 = [A1,ones(Nsample,1)]; % Bias to the next layer
    A_input = [trainX A1_temp3];

    % Clear variables
    clear A1 A1_temp1 A1_temp2 beta1 A1_temp3

end

Training_time = toc;

%% Model Prediction

% Initialization of probability scores
ProbScores = cell(L,1);

% Loop for each output layer
for i = 1:L
    % Generate scores for each class
    A_temp = A{i};
    beta_temp = beta{i};
    trainY_temp = A_temp*beta_temp;

    % Softmax to generate probabilites
    trainY_temp1 = bsxfun(@minus,trainY_temp,max(trainY_temp,[],2));
%for numerical stability
    num = exp(trainY_temp1);
    dem = sum(num,2);
    prob_scores = bsxfun(@rdivide,num,dem);

    % Stores the results
    ProbScores{i} = prob_scores;
end

% Calculate the training accuracy
TrainingAccuracy = ComputeAcc(trainY,ProbScores);

%% Builds trained model
model.L = L;
model.w = weights;
model.b = biases;
model.beta = beta;
model.mu = mu;
model.sigma = sigma;
model.Activation = act_fun;
end
```

➢ File: **MRVFLpredict.m**

```
function [TestingAccuracy,Testing_time] =
MRVFLpredict(testX,testY,model)

% Extract trained model parameters
L = model.L;
w = model.w;
b= model.b;
beta = model.beta;
mu = model.mu;
sigma = model.sigma;
act_fun = model.Activation;

% Initialization
[Nsample,~] = size(testX);
A = cell(L,1);
A_input = testX;

tic
% Loop for every layer
for i = 1:L

    % Hidden Layer Operation
    A1 = A_input * w{i};
    A1 = bsxfun(@rdivide,A1-repmat(mu{i},size(A1,1),1),sigma{i}); %
Standard normalization
    A1 = A1+ repmat(b{i},Nsample,1);                              %
Apply random bias

    % Activation Function
    switch lower(act_fun)
        case 'ReLU' % Range: [0,inf]
            A1 = ReLU(A1);

        case {'sig','sigmoid'} % Range: [0,1]
            A1 = sigmoid(A1);

        case {'sin','sine'} % Range: [-1,1]
            A1 = sin(A1);

        case 'hardlim' % Range: [0,1]
            A1 = double(hardlim(A1));

        case 'tribas' % Range: [0,1]
            A1 = tribas(A1);

        case 'radbas' % Range: [0,1]
            A1 = radbas(A1);

        case 'sign' % Range: [-1,1]
            A1 = sign(A1);

        case 'SELU' % Range: [-1,inf]
            A1 = SELU(A1);
```

```
        otherwise
            error('Activation function not recognized.');

    end

    % Output to classifier layer
    A1_temp1 = [A_input,A1,ones(Nsample,1)];     % Direct links +
bias to the output layer
    A{i} = A1_temp1;

    % Output to next hidden layer
    A1_temp3 = [A1,ones(Nsample,1)]; % Bias to the next layer
    A_input = [testX A1_temp3];

    % Clear variables
    clear A1 A1_temp1 A1_temp2 w1 b1
end

%% Model Prediction

% Initialization of probability scores
ProbScores = cell(L,1);

% Loop for each output layer
for i = 1:L
    % Generate scores for each class
    A_temp = A{i};
    beta_temp = beta{i};
    testY_temp = A_temp*beta_temp;

    % Softmax to generate probabilites
    trainY_temp1 = bsxfun(@minus,testY_temp,max(testY_temp,[],2));
%for numerical stability
    num = exp(trainY_temp1);
    dem = sum(num,2);
    prob_scores = bsxfun(@rdivide,num,dem);

    % Stores the results
    ProbScores{i} = prob_scores;
end

% Calculate the testing accuracy
TestingAccuracy = ComputeAcc(testY,ProbScores);
Testing_time = toc;

end
```