SOFTWARE REQUIREMENTS CLASSIFICATION USING WORD EMBEDDINGS AND CONVOLUTIONAL NEURAL NETWORKS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Vivian Fong

June 2018

© 2018
Vivian Fong
ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE:	Software Requirements Classification Using Word Embeddings and Convolutional Neural Networks
AUTHOR:	Vivian Fong
DATE SUBMITTED:	June 2018
COMMITTEE CHAIR:	Alexander Dekhtyar, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	David Janzen, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Foaad Khosmood, Ph.D. Professor of Computer Science

ABSTRACT

Software Requirements Classification Using Word Embeddings and Convolutional Neural Networks

Vivian Fong

Software requirements classification, the practice of categorizing requirements by their type or purpose, can improve organization and transparency in the requirements engineering process and thus promote requirement fulfillment and software project completion. Requirements classification automation is a prominent area of research as automation can alleviate the tediousness of manual labeling and loosen its necessity for domain-expertise.

This thesis explores the application of deep learning techniques on software requirements classification, specifically the use of word embeddings for document representation when training a convolutional neural network (CNN). As past research endeavors mainly utilize information retrieval and traditional machine learning techniques, we entertain the potential of deep learning on this particular task. With the support of learning libraries such as TensorFlow and Scikit-Learn and word embedding models such as word2vec and fastText, we build a Python system that trains and validates configurations of Naïve Bayes and CNN requirements classifiers. Applying our system to a suite of experiments on two well-studied requirements datasets, we recreate or establish the Naïve Bayes baselines and evaluate the impact of CNNs equipped with word embeddings trained from scratch versus word embeddings pre-trained on Big Data.

ACKNOWLEDGMENTS

Thanks to:

- Professor Alex Dekhtyar for guiding me through my academic endeavors and sparking my interest in machine learning.
- Thesis-depression buddies Tram, Irene, and Mike. Misery loves company.
- Thesis veterans Nupur, Katie, and Andrew for their counseling and encouragement.
- Shelby and the Johannesens for graciously letting me stay with them during my last visits to SLO.
- Lexus, Jacob, Esha, and Gary for retaining some remnants of my social life.
- Laura, JoJo, and David for being my backbone and personal cheer squad.
- Mom and Dad for forever loving and supporting me, feeding me delicious home-cooked meals and treating me like the spoiled princess I am.
- Every coffee shop I step foot in, laptop in hand, during my entire graduate career.

TABLE OF CONTENTS

Page

LIST OF TABLES ix

LIST OF FIGURES xi

CHAPTER

1 Introduction 1

2 Background 5

2.1 Requirements Specifications 5

2.1.1 Requirements Classification 5

2.2 Machine Learning 7

2.2.1 Naïve Bayes 9

2.2.2 Perceptrons 10

2.2.3 Support Vector Machines 10

2.3 Deep Learning 12

2.3.1 History 12

2.3.2 Neural Networks 13

2.3.3 Convolutional Neural Networks 16

2.4 Natural Language Processing 20

2.4.1 Text Preprocessing 20

2.4.2 Vector Representations 21

2.5 Model Validation 25

2.5.1 Cross-Validation 25

2.5.2 Overfitting 25

2.6 Performance Metrics 26

2.6.1 Confusion Matrix 26

2.6.2 Accuracy 27

2.6.3 Recall, Precision, and F_β-score 27

2.6.4 Cross-Entropy Loss 28

2.7 Tools 28

2.7.1 Natural Language Processing Toolkit 28

2.7.2 Scikit-Learn 29
2.7.3 Gensim 29
2.7.4 TensorFlow 29
2.8 Datasets 30
2.8.1 Security Requirements (SecReq) 30
2.8.2 Quality Attributes (NFR) 32
2.9 Prior Work 34
2.9.1 Classifying Security Requirements 34
2.9.2 Classifying Quality Attributes 35
3 Design and Implementation 39
3.1 Run Configurations 41
3.2 Data Loading and Preprocessing 42
3.2.1 Raw Data Formats 42
3.2.2 Classifier Input Format 44
3.2.3 Training and Test Set Division 45
3.2.4 Pre-trained Embedding Model Loading 45
3.3 Classifier Training 46
3.3.1 CNN 46
3.3.2 Naïve Bayes 50
3.3.3 Testing and Result Compilation 50
3.4 Exportation 51
4 Experiments and Results 52
4.1 Experiment 1: Naïve Bayes Baselines 54
4.1.1 SecReq-SE 54
4.1.2 NFR-NF 55
4.1.3 NFR-Types 55
4.1.4 NFR-SE 56
4.2 Experiment 2: Optimal CNN Models for Binary Classification 56
4.2.1 SecReq-SE 59
4.2.2 NFR-NF 60
4.2.3 NFR-SE 60
4.2.4 Discussion 61

4.3 Experiment 3: Optimal CNN Models for Multi-label Classification . . 62
4.3.1 NFR-Types: Multi-label 63
4.3.2 NFR-Types: Binary CNNs 64
4.3.3 Discussion 64
4.4 Experiment 4: Epoch Convergence 65
4.4.1 SecReq-SE 65
4.4.2 NFR-NF 68
4.4.3 Discussion 69
4.5 Experiment 5: Cross-Dataset Security Requirements Classification . . 69
4.5.1 Train on SecReq-SE, Validate on NFR-SE 70
4.5.2 Train on NFR-SE, Validate on SecReq-SE 71
4.5.3 Hybrid Security Dataset 72
4.5.4 Discussion 73
5 Related Work 74
5.1 Preprocessing Strategies 74
5.1.1 Comparison 75
5.2 Sampling Strategies 76
5.2.1 Comparison 77
5.3 Domain-Independent Model and One-Class SVMs 78
5.3.1 Comparison 79
5.4 Topic Modeling, Clustering, and Binarized Naïve Bayes 80
6 Conclusions and Future Work 81
6.1 Future Work 84
6.1.1 One-vs.-All Classification 84
6.1.2 CNN Optimization 84
6.1.3 Word Embeddings 85
6.1.4 FastText 86
BIBLIOGRAPHY 87
APPENDICES
A TF-IDF Experiments 93

LIST OF TABLES

Table Page

2.1 Examples of functional and non-functional requirements from the NFR dataset. 6

2.2 Confusion matrix legend. 26

2.3 SecReq dataset, broken down by SRS [25]. 30

2.4 Examples of security-related and not security-related requirements from SecReq. 31

2.5 NFR dataset, broken down by project and requirements type [14]. . 32

2.6 Examples of requirements of different types from NFR. 33

2.7 SecReq Naïve Bayes classification experiments by Knauss et al. [26]. 35

2.8 Results from using top-15 terms at classification with threshold value of 0.04, by Cleland-Huang et al. [14]. 37

3.1 Run configuration options. 42

3.2 `WordEmbeddingCNN` hyperparameters. 47

3.3 Embedding matrix example ($d = 5, n = 7, N = 10$). 48

4.1 SecReq and NFR primary classification problems. 52

4.2 Naïve Bayes results for SecReq-`SE` binary classification. 54

4.3 Naïve Bayes (with `CountVectorizer`) results for NFR-`NF` binary classification. 55

4.4 Naïve Bayes (with `CountVectorizer`) results for NFR-Types multi-label classification and individual NFR label binary classification. . 56

4.5 SecReq-`SE`: Optimal CNN model results, trained with 140 epochs. . 59

4.6 SecReq-`SE`: Loss and F_1-score from optimal CNN models applied to each word embedding type. 59

4.7 NFR-`NF`: Optimal CNN model results, trained with 140 epochs. . . 60

4.8 NFR-`NF`: Loss and F_1-score from optimal CNN models applied to each word embedding type. 60

4.9 NFR-`SE`: Optimal CNN model results, trained with 140 epochs. . . 61

4.10 NFR-`SE`: Loss and F_1-score from optimal CNN models applied to each word embedding type. 61

4.11 NFR-Types: Optimal multi-label CNN model results, trained with 140 epochs. 63
4.12 NFR-Types: Individual binary CNN results, trained with 140 epochs. 64
4.13 Results from NB and CNN classifiers trained on SecReq-`SE`, validated on NFR-`SE` using 10-fold cross-validation. 70
4.14 Results from NB and CNN classifiers trained on SecReq-`SE`, validated on NFR-`SE`, without cross-validation. 70
4.15 Results from NB and CNN classifiers trained on NFR-`SE`, validated on SecReq, using 10-fold cross-validation. 71
4.16 Results from NB and CNN classifiers trained on NFR-`SE`, validated on SecReq, without cross-validation. 72
4.17 Results from NB and CNN classifiers trained and validated on security hybrid dataset, using 10-fold cross validation. 73

5.1 NFR-`NF` binary classification results from Abad et al. [9] compared to our CNN word embedding approaches. 76
5.2 NFR-`US` and NFR-`PE` binary classification results from Kurtanović et al. [29] compared to our CNN word embedding approaches. . . . 78
5.3 Comparison of most effective security requirements classifiers on single domain evaluations between Knauss et al. [26], Munaiah et al. [37], and our CNN word embedding approaches. 79

6.1 Summary of NB baseline and CNN performance for binary classification problems, trained with 140 epochs and run with 10-fold cross-validation. 82
6.2 Summary of NB baseline and CNN F_1-scores for NFR-Types multi-label classification, trained with 140 epochs and run with 10-fold cross-validation. 82
6.3 Summary of NB baseline and CNN TP count for NFR-Types multi-label classification, trained with 140 epochs and run with 10-fold cross-validation. 83

A.1 Naïve Bayes (with `TfidfVectorizer`) results for SecReq-`SE` binary classification. 93
A.2 Naïve Bayes (with `TfidfVectorizer`) results for NFR-`NF` binary classification. 93
A.3 Naïve Bayes (with `TfidfVectorizer`) results for NFR-Types multi-label classification. 93

LIST OF FIGURES

Figure | Page

2.1 Example of a support vector machine for the linearly non-separable case. 11

2.2 Example of a neural network with three hidden layers. 14

2.3 CNN architecture for image classification (adapted from LeCun et al. [31]). 17

2.4 CNN architecture for sentence classification (adapted from Zhang et al. [45]). 19

2.5 Skip-gram model (adapted from Mikolov et al. [34]). 23

2.6 Recall and precision results from one-vs.-all multi-label classification of NFR types from Casomayor et al. [11]. 38

3.1 System design activity diagram. 40

3.2 Examples of SecReq raw data format. 43

3.3 Example of NFR raw data format. 43

3.4 Illustration of `WordEmbeddingCNN` layer initialization. 46

4.1 # of Epochs vs. Performance for SecReq-`SE` optimal CNN. 66

4.2 # of Epochs vs. Performance for NFR-`NF` optimal CNN. 67

Chapter 1

INTRODUCTION

Requirements engineering (RE) describes the process of discovering, documenting, and maintaining requirements during the software development life cycle [43]. Requirements outline the business needs and use cases that establish the necessity of a product, as well as overall system performance criteria. Whether it be for a school assignment or industry development, the modeling and fulfillment of requirements is crucial to measuring a product's completion and a team's success. Design and implementation decisions should all directly correlate with requirements established within the project.

Software requirements can be of different types, encapsulating criteria within a specific area of interest in the software product. For example, requirements can be classified as functional (explicit features, or *functions*, of the product) or non-functional (implicit quality criteria for the product), which can be further drilled into more specific categories. Furthermore, requirements may serve different purposes, from highlighting security vulnerabilities to measuring scalability necessities to assessing general look-and-feel [4, 14]. Identifying all requirements of a specific type (i.e., security-related) allows engineers and other participants of the software development cycle to hone in on particular non-functional concerns for the system and assess project completeness, ultimately promoting awareness of requirements that are often overlooked. Software specialists can immediately locate which requirements interest them without needed to peruse through the entire SRS (e.g., the UX designer is likely interested in look-and-feel requirements). However, the manual task of labeling what category a requirement falls under is tedious. On top of that, manual requirements labeling requires domain-expertise, which can be limited and expensive, highlighting

the need to explore automated methods.

Automated requirements labeling can be defined as a machine learning classification problem. Machine learning is a sub-field within artificial intelligence that encompasses a type of algorithm that discovers, or *learns*, patterns from existing data, detecting trends to help make predictions on new data [36]. Classification is the machine learning task of identifying which category out of a set of categories an item belongs to [19]; it requires a set of pre-labeled data to learn from, using that knowledge to predict the labels for unseen data. Previous automated requirements classification research primarily investigate traditional learning and vectorization methods such as Naïve Bayes and TF-IDF.

In this thesis, we aim to study the impact of deep learning, a subdivision within machine learning, on the classification of software requirements. Deep learning techniques, characterized by multi-layer graphs of data transformation, are booming in popularity with their breakthroughs in machine translation, image and voice recognition, and other fields within technology. In recent years, deep learning has helped develop a way to transform text into a medium that computers can consume and extract semantic information from, making advancements towards replicating the human ability to process language. At their core, requirements specifications are plain text documents that can be processed like natural language. They are commonly very short in length and written in formal language with domain-specific diction. In addition, requirements specifications contain a relatively small volume of samples. These characteristics foster an unconventional environment for deep learning as such methods are often applied on extremely large datasets of feature-rich samples.

We investigate two specific aspects of deep learning: (1) convolutional neurals networks to train a classifier in performing requirements classification, and (2) word embeddings to represent our requirements documents. A convolutional neural net-

work (CNN) is a deep neural network designed to learn from a grid-like topology in the input data [19]. Traditionally, CNNs are used to tackle image recognition tasks, but its efficacy in text classification has been recently proven [23]. Word embeddings are rich vector representations of words that claim to capture syntactic and semantic relationships between words, resultant from training neural networks on very large corpora (*"Big Data"*) [35, 34]. This leads us to pose our primary research questions:

1. **RQ1:** Can deep learning models such as CNNs offer competitive performance on software requirements classification?

2. **RQ2:** Can leveraging the power of Big Data when vectorizing our documents with pre-trained word embeddings boost CNN performance on software requirements classification?

Our paper to the 25th International IEEE Requirements Engineering Conference titled *"RE Data Challenge: Requirements Identification with Word2Vec and TensorFlow"* [17] initiates our research with a replication of prior work baselines in addition to a pilot assessment of word2vec word embeddings and TensorFlow CNNs on two binary requirements classification problems. Since that paper, we dive deeper into the study to assemble the following list of contributions:

1. Recreation or establishment of Naïve Bayes baselines.

2. Feasibility assessment of CNNs on binary and multi-label requirements classification.

3. Comparison of three word embedding methods in assisting requirements document representation when training CNNs.

4. A set of evaluations of requirements classification using two well-studied datasets.

The rest of this document is organized as follows: Chapter 2 dives into the background in software requirements engineering, machine learning methodology and tools, as well as the datasets we utilize and some prior work. Chapter 3 details our system design, and Chapter 4 outlines our experiments and discusses their results. Chapter 5 briefly discusses the current related work active in the field. Lastly, Chapter 6 summarizes our conclusions and future work.

Chapter 2

BACKGROUND

2.1 Requirements Specifications

Upon the launch of a product idea, product managers in software teams need to understand the needs of their potential customers and users, a process called *requirements elicitation*. Depending on the development style adopted by the team, the requirements elicitation process may take a long time to complete. The artifacts produced during this process are *software requirements specifications* (SRS), documents that list and detail each user or system requirement that needs to be satisfied for the product to be complete.

Various types of requirements are considered during requirements elicitation. Functional requirements (FR) can define specific behaviors, features, and use cases of the product. FRs can be broken down into high-level requirements (HLR) and low-level requirements (LLR). HLRs can be abstract statements defining an overall feature needed, and LLRs are more detailed descriptions of what the product needs in order to realize the HLR. Non-functional requirements (NFR), on the other hand, assess system properties and constraints such as performance, scalability, and security [43]. In short, FRs describe *what* the system should do and NFRs describe *how* the system should perform it [18]. Table 2.1 showcases some examples of functional and non-functional requirements.

2.1.1 Requirements Classification

Requirements elicitation, SRS documentation, and maintenance all make up a process called *requirements engineering* (RE) [43]. An area of research within requirements

Table 2.1: Examples of functional and non-functional requirements from the NFR dataset.

Requirement Type	*Requirements Text*
Functional	"The system will notify affected parties when changes occur affecting classes including but not limited to class cancellations class section detail changes and changes to class offerings for a given quarter."
Performance	"Any interface between a user and the automated system shall have a maximum response time of 5 seconds unless noted by an exception below."
Scalability	"The product shall be capable of handling up to 1000 concurrent requests. This number will increase to 2000 by Release 2. The concurrency capacity must be able to handle peak scheduling times such as early morning and late afternoon hours."
Security	"User access should be limited to the permissions granted to their role(s) Each level in the PCG hierarchy will be assigned a role and users will be assigned to these roles. Access to functionality within RFS system is dependent on the privileges/permission assigned to the role."

engineering is requirements classification.

Requirements classification (or *requirements identification*) is the task of identifying requirements as belonging to a specific category, thus highlighting their role in the project. Two examples of classification tasks are (1) distinguishing between functional and non-functional requirements, and (2) determining whether a non-functional requirement is related to concerns such as security, performance, reliability, etc [14].

NFRs are important because they pinpoint areas that affect the health and well-being of the system as a whole rather than just a single feature in a module. Fulfilling them requires not only careful design decisions in the beginning, but also continuous effort throughout the entire software development process. Unfortunately, as Kurtanović et al. concludes [29], NFRs are often identified later in the software process [12, 29] and are vaguely described and poorly managed [18], causing engineers to neglect their importance [11]. Classifying requirements can help promote transparency and organization within the SRS and stimulate awareness toward crucial system con-

cerns that should be as much of a priority to engineers as feature development.

However, manually classifying requirements calls for engineers who have expertise in the respective areas (i.e., proper identification of security requirements calls for security knowledge). This resource barrier can discourage project managers and engineers from properly assessing NFRs throughout the development process, leaving neglected or unidentified issues in the back seat and thus amplifying the risk for defects, performance inadequacies, and technical debt. Automating this task can alleviate the need for domain-experts, which in turn can promote the practice within the software community. In order to automate requirements classification, we must explore machine learning methodologies.

2.2 Machine Learning

Machine learning is a sub-field within AI that studies the making of predictions on data by learning from the characteristics of past samples [36]. Machine learning is a form of applied statistics that utilizes computers to estimate extremely complex functions [19], making it a powerful mechanism for solving abstract problems that are too difficult for humans to specify explicit algorithms for [36].

Machine learning algorithms can be divided into two categories: supervised and unsupervised learning. *Supervised learning* is the category of learning algorithms that builds a model by training on data that have been annotated with labels [19]. Having pre-labeled training data provides the model information as to how many classes exist within the data, allowing the model to focus on analyzing the features that distinguish these particular classes instead of formulating groups from scratch. Oftentimes, the requisite for labeled data poses a hurtle as most data in this world are untagged and manual tagging is often impractical. In contrast, *unsupervised learning* algorithms train on unlabeled data [19]. The advantage of unsupervised methods is

their disregard for explicit labeling. In the process, the model generates predictions of the possible distinguishing groups within the data.

Machine learning can be utilized to tackle a variety of problem areas, including regression, classification, and anomaly detection [19]. The work of this thesis focuses on supervised learning approaches to classification problems.

Classification

Classification is the task of determining which category, or *class*, out of k categories an input belongs to [19]. More formally, with an input vector $\mathbf{x}$, a classification algorithm needs to formulate a function $f : \mathbb{R}^n \rightarrow \{1, \ldots, k\}$ so that $y = f(\mathbf{x})$, outputting the predicted class y [19].

A classic example of *binary classification* is the image recognition task of distinguishing whether a photo of a fluffy animal is one of a cat or dog. The input can be represented as a matrix of numerical pixel values $\mathbf{x}$, and the output can be a one-hot vector y signaling which class the image is predicted to belong to. The example becomes a *multi-label classification* problem if we add more animals into the list of possible animal categories.

Numerous different types of learners can perform the task of classification, also known as *classifiers*. In the following subsections, we discuss Naïve Bayes, perceptrons, and support vector machines. Rather than perform a deep dive into the mathematics or algorithms, the purpose of these sections is to provide a brief introduction to the nature and construction of these classifiers.

2.2.1 Naïve Bayes

The Bayesian learner, also known as the *Naïve Bayes* (NB) classifier, is a straightforward probabilistic approach [36]. Despite its simplicity, Naïve Bayes is very practical and its performance can sometimes rival more sophisticated methods [36]. Consequently, NB is often a common first choice when tackling text classification tasks.

As this thesis involves textual requirements, we will explain NB in terms of document analysis. Given categories $C = \{c_1, \ldots, c_{|C|}\}$ and documents $\mathbf{D} = \{\bar{d}_1, \ldots, \bar{d}_{|D|}\}$, the probability P that the document $\bar{d}_j$ belongs to category c_i is computed in Bayes Theorem [41, 36]:

$$P(c_i \mid \bar{d}_j) = \frac{P(c_i)\, P(\bar{d}_j \mid c_i)}{P(\bar{d}_j)} \tag{2.1}$$

where document $\bar{d}_j = \langle w_{1j}, \ldots, w_{|T|j} \rangle$ is represented by a vector of weights for each term in the vocabulary set T from all documents $\mathbf{D}$. The topic of vector representations is further elaborated in Section 2.4.2.

$P(\bar{d}_j)$ is the probability that a random document in the corpus is represented by vector $\bar{d}_j$, and $P(c_i)$ is the probability that a random document in the corpus is of category c_i [36]. Because the number of possible variations for $\bar{d}_j$ is too high, computing $P(c_i \mid \bar{d}_j)$ can be impossible. To alleviate this bottleneck, an assumption that "any two coordinates of the document vector are, when viewed as random variables, statistically independent of each other" is made [41]. This independence assumption is what characterizes this method as a *naïve* approach. Equation 2.2 is the formula for the independence assumption [41, 36]:

$$P(\bar{d}_j \mid c_i) = \prod_{k=1}^{|T|} P(w_{kj} \mid c_i) \tag{2.2}$$

2.2.2 Perceptrons

A *perceptron* is a simple linear binary classifier that determines a function that can bisect linearly separable data into two classes in d-dimensional space [32].

Let $X = \{\bar{x}_1, \ldots, \bar{x}_n\}$ represent the set of data points where each sample $\bar{x}_i$ is represented by a d-dimensional vector $langlea_1, \ldots, a_d\rangle$. With $C = \{+1, -1\}$ as the set category labels, let $Y = \{y_1, \ldots, y_n\}$ where $y_i \in C$ so that y_i is the true class of the input $\bar{x}_i$. Given a vector of *weights* $\mathbf{w} = \langle w_1, \ldots, w_d\rangle$ and threshold value θ, the perceptron function

$$f(\bar{x}) = \mathbf{w} \cdot \bar{x} = \sum_{j=1}^{d} w_j \cdot a_j \tag{2.3}$$

determines the class of input $\bar{x}$ with the following decision procedure:

$$class(\bar{x}) = \begin{cases} +1 & \text{if } f(\bar{x}) > \theta \\ -1 & \text{if } f(\bar{x}) < \theta \end{cases} \tag{2.4}$$

The $f(\bar{x}) = \theta$ case is always counted as a misclassification [32].

Training a perceptron requires iterative fine-tuning of weight vector $\mathbf{w}$ until either (1) the function $f(\bar{x})$ either correctly classifies all $\bar{x} \in X$, or (2) the error rate converges and stops decreasing [32]. Essentially, with each iteration of the training algorithm, the perceptron function tilts and adjusts in the d-dimensional space until it can successfully separate the two classes. Perceptrons are limited to a single linear hyperplane, rendering them useless in cases with non-linear or ambiguous separation boundaries.

2.2.3 Support Vector Machines

A *support vector machine* (SVM) is essentially an improved perceptron that can classify non-linearly separable data [32]. A SVM determines the optimal hyperplane that

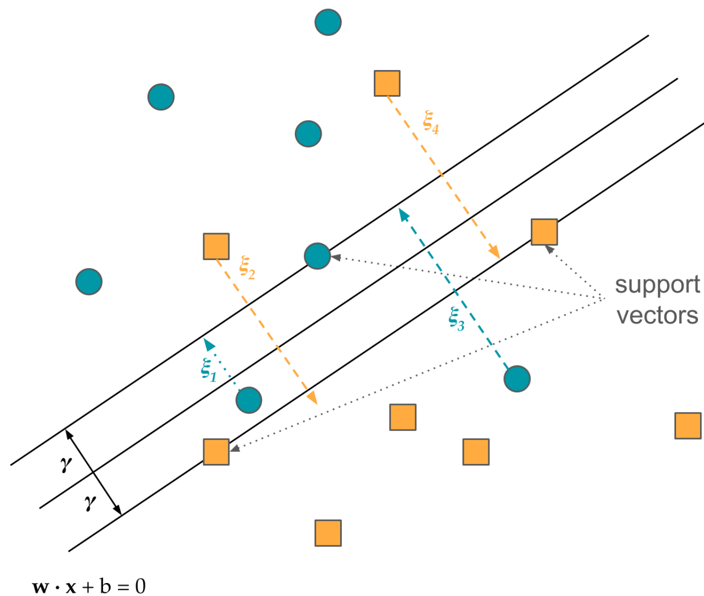


Figure 2.1: Example of a support vector machine for the linearly non-separable case.

divides the two classes of data and maximizes the distance between the hyperplane and the closest data samples [32, 44, 41]. Adopting the same variable definitions as Section 2.2.2, the function for the hyperplane is defined as follows [44]:

$$h(\bar{x}) = \mathbf{w} \cdot \bar{x} + b = (\sum_{j=1}^{d} w_j \cdot a_j) + b = 0 \tag{2.5}$$

where b is a constant scalar value called the *bias* which can be treated the same as the negative of the threshold θ in our discussion of perceptrons [32, 44].

The data points with the shortest distance γ (also referred to as the *margin*) to the hyperplane are known as *support vectors*. The goal of the SVM is to determine the weight vector $\mathbf{w}$ and bias b that maximize γ such that for all $i = 1, \ldots, n$, the following condition is fulfilled [32]:

$$y_i \cdot (\mathbf{w} \cdot \bar{x}_i + b) \geq \gamma \tag{2.6}$$

Figure 2.1 illustrates a two-dimensional, linearly non-separable example of a SVM that separates the squares from the circle samples. For this case, there are samples

of one class that on the wrong side of the hyperplane. To address this, Equation 2.6 can be revised by introducing *slack variables* ξ_i [32]:

$$y_i \cdot (\mathbf{w} \cdot \bar{x}_i + b) \geq 1 - \xi_i \,, \quad \xi_i > 0 \tag{2.7}$$

Penalties can be added (with the *hinge loss* function) to account for data points that might be on the wrong side of the hyperplane [32]. SVMs can also use *kernal tricks* to build non-linear separating curves.

2.3 Deep Learning

Deep learning is a subcategory of machine learning that solves a complex problem by learning from a hierarchy of smaller, simpler representations, building a *deep* graph of concepts with many layers [19].

2.3.1 History

Deep learning has a long train of history dating back to the 1940s, adopting several aliases and riding waves of different philosophies throughout the decades. Although the methodology is a current hot topic, deep learning has remained dormant and unpopular for most of its history [19].

The first wave was introduced with the study of cybernetics in biological learning and implementations of the perceptron in the 1940s–1960s. The second wave came between 1980–1995 with the concept of backpropagation and neural network training. Finally, the third and current wave arrived in 2006, adopting the buzzword *deep learning* that we know and love today. Today's appreciation for deep learning is thanks to modern day computing infrastructure and growing data availability [19, 31], allowing researchers and industry to utilize the science with their massive volumes of data, contributing significant advancements in AI.

2.3.2 Neural Networks

The fundamental example of a deep learning model is the *deep feedforward neural network*, a concept loosely inspired by the shape and nature of neural connectivity within the brain. A neural network estimates a mathematical function f^* mapping some input $\bar{x}$ to some class label y, composed of a web of simpler intermediate functions [19]. The model is also known as the *multi-layer perceptron* (MLP) as it can be seen as an "acyclic network of perceptrons", with the output of some perceptrons used as the input to others [32]. Each individual perceptron in the network is known as a *neuron*, with the fundamental difference being the use of a non-linear activation function rather than a unit-step decision function. The following subsections describe the architecture and training of a neural network.

Architecture

In a feedforward neural network, the input $\bar{x}$ flows through the network of functions to reach an output $\hat{y}$ [19].

As shown in Figure 2.2, the initial layer to the network is the *input layer*, representing the units of the input vector $\bar{x}$. The subsequent layers are known as *hidden layers*. If the neural network $f(\bar{x})$ is composed of three chained functions such that $f(\bar{x}) = f_3(f_2(f_1(\mathbf{x})))$, f_1 is the first hidden layer of the network, f_2 the second, and f_3 the third. These layers are "hidden" because the output of each function f_i are unknown to the input data. Finally, the last layer of a feedforward network is the *output layer*, which represents the class label $\hat{y}$ concluded by the classifier. The length of the chain determines the *depth* of the network (hence "deep" learning).

Each neuron in its layer works in parallel. A neuron receives a vector of inputs and weights from the previous layer to serve as an input to its *activation function*,

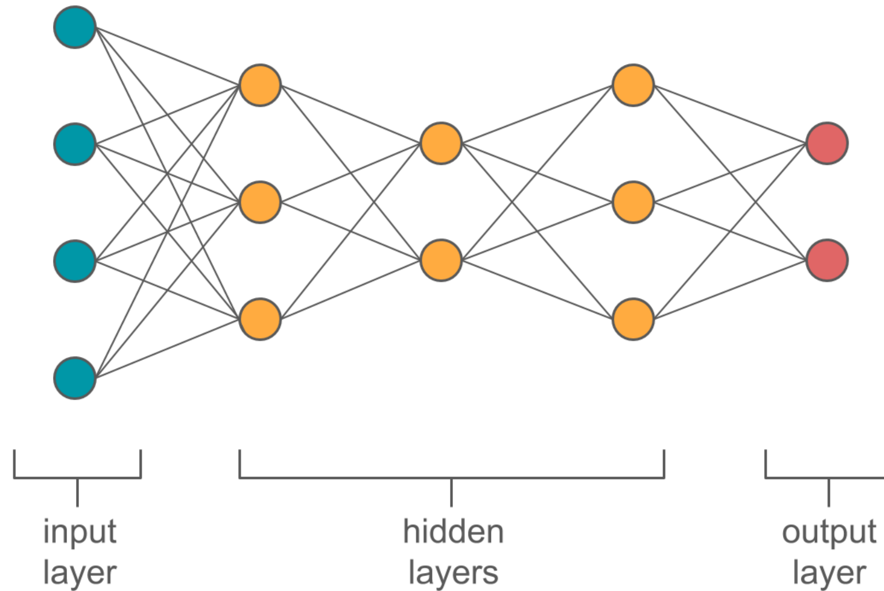


Figure 2.2: Example of a neural network with three hidden layers.

a non-linear sigmoid function g (i.e., $\sigma(x)$, $tanh(x)$, ReLU) that we choose when designing the model [19]. In other words, the output of a neuron is $g(\sum w_i x_i)$, the activation function applied on the sum of the scalar product between the weight and input vectors from the previous layer, which is then sent over to the next layer as an input.

In the case of SVMs and neural networks, we need to select a *cost function* that represents the error of the model $f(\mathbf{w}, b)$ to optimize. Standard cost functions include sum-squared error and cross-entropy loss [19]. Cross-entropy loss is discussed in Section 2.6.4.

Gradient Descent

Gradient descent is a useful iterative approximation technique employed to train many types of machine learning models. It looks for the optimal value for multivariate functions, computing the gradient of the function and tuning the parameters with each learning step to approach a local optima [32].

Let η be the *learning rate*, or the fraction of the gradient we move $\mathbf{w}$ by in each round. During each iteration, each $w_j \in \mathbf{w}$ gets adjusted by the following formula [32]:

$$w_j := w_j - \eta \nabla f(w_j) \tag{2.8}$$

It is evident that the learning rate η can play a large role in how quickly or accurately gradient descent can perform; the weights might take too long to converge if η is too small, whereas if η is too large, the optimum can be missed. Optimization algorithms, such as *Adam* [24], have been developed to provide sophisticated strategies to optimizing the learning rate and the performance of gradient descent.

Training

The outline below describes the steps to training a neural network [19]:

1. **Initialization.** Initialize the weights $\mathbf{w}$ for each neuron in the hidden and output layers.

2. **Learning Step.** Each learning step is performed in two stages:

   - *Forward Propagation.* On each step s, propagate a *batch* of input points $X_s \subseteq X$ through the network and compute the cost of the model with the current weights.

   - *Back Propagation.* Apply gradient descent on the current batch X_s from the output layer through the hidden layers to adjust the weights of the neurons.

3. **Termination.** Terminate once either the cost converges or drops below a certain threshold.

Hyperparameters

Hyperparameters are settings for the classifier that need to be determined before training [13, 19]. In the case of neural networks, the number of hidden layers and the number of neurons per layer , learning rate, and activation function are examples of hyperparameters that need to be set prior to training. The selection of hyperparameters can greatly affect the performance of the model, however finding the optimal combination of settings is proven to be an ongoing challenge in machine learning [13].

2.3.3 Convolutional Neural Networks

The *convolutional neural network* (CNN) is an evolution of the multi-layer perceptron that specializes in automatic feature extraction [42] from data that can be processed in a "grid-like topology" [19]. In other words, CNNs are designed to take advantage of the locality and order of the input elements when learning, making it compatible with tasks involving pattern recognition [17, 31]. The following subsection describes the general architecture of a CNN for image recognition, followed by a deeper discussion of a CNN model designed for sentence classification.

CNN Architecture

CNNs are composed of a stack of three types of layers: convolutional, pooling, and fully-connected layers [38, 19]. Figure 2.3 illustrates an example architecture for a CNN meant to classify handwritten digits from the MNIST dataset [31].

Input Layer. Just like with traditional neural networks, the *input layer* for a CNN takes in a vector of input values. In the case of the image processing, the input is a $n \times n$ matrix of pixel values representing the image. The input can also be expanded in dimensionality to encompass multiple channels of values per unit in the grid (e.g.,

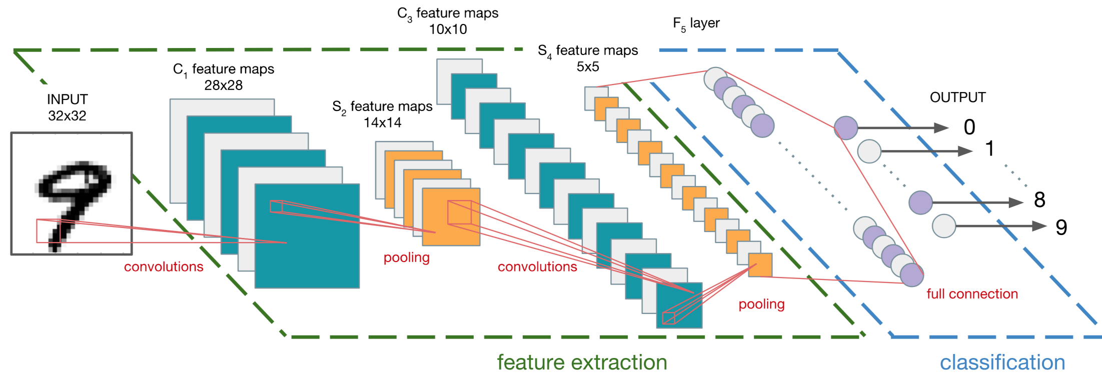


Figure 2.3: CNN architecture for image classification (adapted from LeCun et al. [31]).

RGB values per pixel).

Convolutional Layer. In the *convolutional layer*, a *filter* of size $h \times h$ is used to slide across the input matrix, capturing $n - h + 1$ regions of the input called *convolutions*. Each convolution acts as a neuron, computing the scalar product between its weights and regional input, followed by the activation function (namely *rectified linear unit*, or ReLU) [38], resulting in a single feature. The features from each convolution are aggregated into a *feature map* for each filter.

Pooling Layer. The *pooling layer* reduces the dimensionality of its input by a process called *subsampling* [38, 19]. An example of subsampling is *max-pooling* where the greatest value from the neural outputs from the convolutions from a particular region is taken and the rest are discarded.

Fully-Connected Layers. The *fully-connected layers* that follow perform the same operations as traditional neural networks — calculating scores to derive a prediction of which class the input belongs to [38, 31]. A *softmax layer* is often used as the final layer to a neural network, using the softmax function to normalize the output from

the final hidden layer into probability values for each output class [45, 23]. *Dropout* can also be applied to the softmax layer as a means of regularization — randomly "turning off" neurons in the network by randomly settings values in the weight vector to 0 in efforts to prevent overfitting [45]. The concept of overfitting is further discussed in Section 2.5.2.

Through a combination of convolutional and pooling layers, CNNs transform the complexity of the original input data, extracting the core patterns that distinguish one class from another from the training data.

One-Layer CNN for Sentence Classification

Although originally designed for image recognition [28], CNNs have also been proven to be effective textual contexts as well [15, 23]. We will discuss the one-layer CNN architecture Kim et al. has designed for sentence classification [23], illustrated in Figure 2.4.

Embedding Layer. We refer to the input layer in a sentence classification CNN as the *embedding layer*, as the input sentence is formulated as a two-dimensional *embedding matrix* built by concatenating together d-dimensional word vector representations for each of the n tokens in the sentence [23, 45]. Figure 2.4 showcases an example where the input sentence `"This was a terrible disappointment!"` is transformed into an embedding matrix with 5-dimensional word embeddings. Word embeddings are further discuss in Section 2.4.2.

Convolutional and Pooling Layers. The *convolutional layer* for a sentence classification CNN uses filters of size $h \times d$ to build $n - h + 1$ convolutions of the input, representing n-grams of size h within the sentence [23, 45]. An *n-gram* is a continuous sequence of n tokens in a sample of text. These n-grams are analogous to the regions

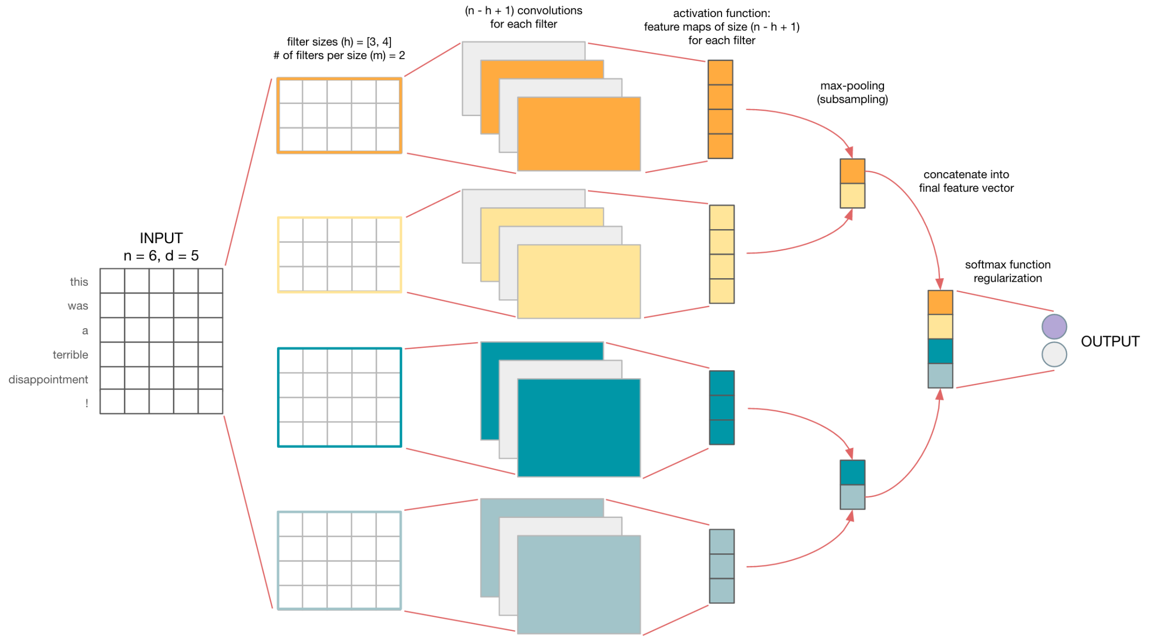


Figure 2.4: CNN architecture for sentence classification (adapted from Zhang et al. [45]).

of pixels in an image, hoping to capture location-based features within the input.

Following the convolutional layer is a *pooling layer* where subsampling is performed to record the best outcome from each filter's feature map. Max-pooling is often employed to capture the most important feature from each feature map [23, 45]. The selected outputs from each feature map are then concatenated into a single feature vector, which is then funneled into the remaining layers of the network.

Although we allude in the earlier discussion of image classification CNNs that the architecture can support a series of convolutional and pooling layers, Kim et al. designed their sentence classification CNN with a single convolutional and pooling layer [23]. The single convolutional layer can support a number of different filter sizes, thus capturing various forms of n-grams. Figure 2.4 illustrates an example with two filter sizes 3×5 and 4×5 ($h_1 = 3, h_2 = 4$), representing 3-grams and 4-grams. The example features two filters of each size ($m = 2$), resulting in $m \cdot (n - h_1 + 1) = 8$

convolutions of size 3×5 and $m \cdot (n - h_2 + 1) = 6$ convolutions of size 4×5. The results from each convolution aggregate into a feature map of shape $(n - h + 1) \times 1$ for each filter.

Fully-Connected Layers. The remainder of the network is identical to the image processing CNN; the network needs to convert the feature vector from the pooling layer into class prediction scores (i.e., softmax layer with optional regularization methods).

2.4 Natural Language Processing

Natural language processing (NLP) studies the interpretation and generation of natural language by computers [21].

2.4.1 Text Preprocessing

Text preprocessing, a pipeline of cleaning operations to transform the raw, free-form text into a "well-defined sequence of linguistically meaningful units" [21], is an essential step for any NLP task. Real-world natural language documents (or *corpora*) are often littered with typos, noise, as well as complex sentence patterns and diction variation. Although NLP is relevant to all languages, the following subsections explore standard preprocessing techniques told in the perspective of processing English text.

Text Segmentation

The first step in the preprocessing pipeline is often *text segmentation*, the process of converting a corpus into sentence or word components. Word segmentation (or *tokenization*) is often the most granular operation, splitting the text into individual

word units known as *tokens* [21, 33]. This involves defining the boundaries for a word, separating tokens by whitespace and punctuation as well as splitting contractions.

Text Normalization

Text normalization describes the standardization of linguistic variation. This ranges from simple heuristics such as case normalization and stemming, to more complex lexical analysis like lemmatization [21, 33].

Stemming refers to the heuristic process of chopping off the end of a word in hopes to remove derivational affixes [33]. *Lemmatization* refers to the process of normalizing morphological variants of the words in a corpus [21, 33]. For example, in a lemma dictionary, the set of verbs

$$B = \{\texttt{"see"}, \texttt{"saw"}, \texttt{"seeing"}, \texttt{"seen"}\}$$

all map to the same verb `"see"` [33]. This implies that all encounters of verbs in set B can be converted to their lemma `"see"`.

Stop Word Removal

Stop words are very common words that provide little to no value in the processing of a corpus (e.g., `"a"`, `"the"`, `"to"`) [33]. Sometimes the removal of stop words can reduce noise in a document and improve NLP performance.

2.4.2 Vector Representations

In order to extract features from documents to carry out classification, the text needs to be converted to some form of vector representation that provides quantitative characteristics of the text.

Frequency

The most basic representation is word count vectors, where each document in the corpus is represented by a vector of the total vocabulary size marking frequencies for each word in the document.

TF-IDF

A vector representation is *term frequency–inverse document frequency* (TF-IDF) [40, 41], often used in information retrieval and data mining [33]. A TF-IDF vector is formed by calculating the TF-IDF weight for each term t in document d by multiplying its term frequency $tf_{t,d}$ and inverse document frequency idf_t:

$$tf_{t,d} = f_{t,d} \tag{2.9}$$

$$idf_t = \log \frac{N}{df_t} \tag{2.10}$$

$$tfidf_{t,d} = tf_{t,d} \cdot idf_t \tag{2.11}$$

where $tf_{t,d}$ measures the frequency of term t in document d, and idf_t measures the importance of t relative to its document frequency df and the total number of documents N [40, 41].

Word2vec

Google introduced *word2vec* [8] in 2013, a toolkit of model architectures to train to produce word vector representations that can retain linguistic contexts that are lost in previous vectorization methods. Supported by Google's ever-growing supply of online corpora, Mikolov et al. designed a collection of shallow neural networks to learn "high-quality distributed vector representations that capture a large number of precise syntactic and semantic word relationships" [35, 34]. These word vector representations, now coined as *word embeddings*, can feature several hundred dimensions.

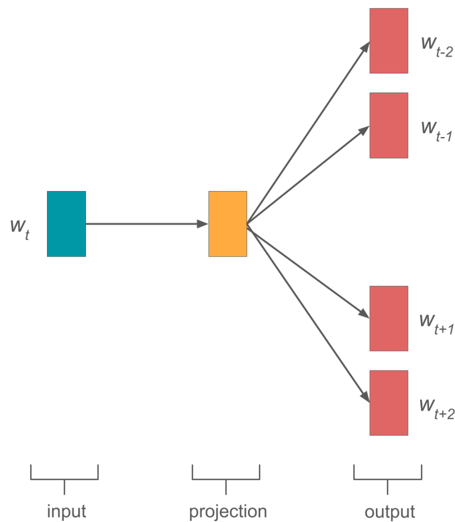


Figure 2.5: Skip-gram model (adapted from Mikolov et al. [34]).

Skip-gram Model. Mikolov et al. designed the *Skip-gram model* in attempt to train a shallow neural network to predict neighboring words to a word in a corpus [34]. Although the outputs from this model were unsuccessful at answering their research question, they noticed interesting attributes to the penultimate layer in the network. The penultimate layer in the Skip-gram model contained vectors later dubbed as word embeddings, yielding word representations in a multidimensional space.

Use Cases. The key piece of novelty to word2vec is its ability to encapsulate not just syntactic similarity, but also semantic relationships between words with the use of standard vector arithmetic. For example,

$$vector(\texttt{"king"}) - vector(\texttt{"man"}) + vector(\texttt{"queen"}) \approx vector(\texttt{"woman"}) \tag{2.12}$$

On the same wavelength, word2vec can discern that `"France"` is to `"Paris"` as `"Germany"` is to `"Berlin"` and other similar relationships [34]. These examples showcase how the advancements of word2vec propels NLP research several steps toward the ultimate goal of supplying computers the ability to decode natural language like humans can.

While word2vec provides the tools to build your own word embeddings from a text corpus, Google has also released a pre-trained word2vec model trained on the 100 billion word Google News corpus, producing 3 million 300-dimensional word embeddings [8].

FastText

Open-sourced by Facebook AI Research (FAIR) lab in 2016, *fastText* [1] is a library that employs state-of-the-art NLP and machine learning concepts. FastText presents two major contributions: (1) a text classifier that drastically improves computational efficiency from neural network models [22], and (2) an advancement to the word2vec Skip-gram model in training word embeddings [10].

As fastText was released recently and caught our attention toward the tail-end of our research, we were only able to incorporate the embeddings into our experiments.

Subword Model. The Skip-gram model represents each word with one distinct vector representation, ignoring the internal structure of words [10]. This is a limitation that is exceptionally relevant to morphologically rich languages, such as Turkish and Finnish. In response, Bojanowski and Grave et al. propose the *Subword model* where every word is represented by bag of character n-grams (i.e., $3 \leq n \leq 6$) of the word plus the word itself [10]. The vector representation for a word is then calculated as the sum of all the vector representations for its n-grams. The Subword model extension to Skip-gram has proven to build more accurate vector representations for complex, technical, and infrequent words.

FAIR lab has published 294 pre-trained fastText models trained on different language versions of Wikipedia [1]. The English version contains 1 million 300-dimensional word embeddings.

2.5 Model Validation

In order to fairly assess the performance of a model on a dataset, the classifier needs to be trained and evaluated on different partitions of the dataset. The portion used for training is known as the *training set*, whereas the remainder of the dataset used for testing is known as the *test set* or *validation set* [19]. A common training-test set ratio is 80% for training and 20% for test.

2.5.1 Cross-Validation

Statistical uncertainty can arise for small validation sets, as a single validation set might not properly represent the dataset as a whole. A technique to counteract this dilemma is *k-fold cross-validation*: repeating the training and testing procedure on k randomly selected, non-overlapping partitions of the dataset and taking the average score from all k folds [27, 19]. A common choice for k is 10, resulting in ten validation trials. This strategy ensures that the classifier gets a chance to train and test on different portions of the dataset, reducing the influence of unique characteristics that might not be representative of the entire dataset.

Stratification. An accessory on top of cross-validation is *stratification*, a process where each fold is engineered to contain approximately the same ratio of classes as the whole dataset, ensuring a decent representation of the original dataset in every fold [27].

2.5.2 Overfitting

A major challenge in training machine learning models is *overfitting*, a condition where the model is learning too many features specific to the training data, missing

big-picture trends [19]. We train machine learning models to ultimately use their predictive power on new, unseen data in the future, so an overfitted model is deemed rather useless as it cannot recognize the same patterns in a different dataset.

2.6 Performance Metrics

Machine learning tasks often employ a suite of metrics to gauge the performance of classifiers. The subsections below define the measures we consider throughout our work.

2.6.1 Confusion Matrix

A *confusion matrix* is often used for binary classification tasks, showcasing how well the items in a validation set are classified and providing more details on the performance of the classifier. Table 2.2 displays the different labels a class prediction can take, given the status between the true value and the predicted value.

Table 2.2: Confusion matrix legend.

		True Value	
		Positive	*Negative*
Predicted Value	*Positive*	TP	FP
	Negative	FN	TN

True positives (TP) are positively-labeled samples that are correctly predicted as positive. *False positives* (FP) are negatively-labeled samples that are incorrectly predicted as positive. *True negatives* (TN) are negatively-labeled samples that are correctly predicted as negative. *False negatives* (FN) are positively-labeled samples that are incorrectly predicted as negative.

2.6.2 Accuracy

Accuracy is the percentage of correctly classified samples overall. If N is the size of the validation set, then

$$accuracy = \frac{TP + TN}{N} \tag{2.13}$$

Accuracy is a primitive measure as it does not tell us exactly how well a model is at classifying a specific class. For example, if the validation set has three positive samples and seven negative samples, and the classifier predicts all ten samples to be negative, then it achieves a seemingly decent accuracy of 70%. However, upon closer inspection, the model classified everything as negative and failed to gather features distinguishing the two classes, making it a weak classifier.

2.6.3 Recall, Precision, and F_β-score

To counteract the inadequacies of the accuracy measure, machine learning studies often supplement their metrics with recall, precision, and their harmonic mean. The following definitions describe the metrics in terms of classifying the positive class.

Recall is the percentage of positively-labeled samples that are successfully predicted:

$$recall = \frac{TP}{TP + FN} \tag{2.14}$$

Precision is the percentage of positively predicted samples that are actually labeled positive:

$$precision = \frac{TP}{TP + FP} \tag{2.15}$$

F_β-score is the weighted harmonic mean of recall and precision, where β measures the relative importance of the two:

$$F_\beta = (1 + \beta^2) \cdot \frac{recall \cdot precision}{\beta^2 \cdot recall + precision} \tag{2.16}$$

When $\beta = 1$, the measure does not weigh preference to either recall or precision, meaning F_1-score is best when $recall = precision = 1$ and worst when $recall = precision = 0$.

2.6.4 Cross-Entropy Loss

Cross-entropy loss, or just *loss*, is the negative log-likelihood between the training data and the model distribution. More formally, cross-entropy loss L measures how close the probability distribution between the true distribution p and the predicted distribution q [19]:

$$L(p, q) = -\sum_{x} p(x) \cdot \log\, q(x) \tag{2.17}$$

Cross-entropy loss is a common cost function used to assess the performance of neural networks.

2.7 Tools

To assist in our research endeavors, we fortunately have access to well-developed open-source tools for document processing, embedding management, classifier construction, and model training.

2.7.1 Natural Language Processing Toolkit

Natural Language Processing Toolkit (NLTK) [3] is exactly what its name implies — a Python toolkit for NLP tasks. NLTK provides access to over 50 corpora and lexical resources as well as libraries for text processing operations such as tokenization, stemming, and lemmatization [3].

2.7.2 Scikit-Learn

Scikit-Learn [5] is a well-established machine learning package available for Python programs. The library includes a rich suite of machine learning implementations, allowing us to employ their versions of traditional classifiers such as Naïve Bayes. Scikit-Learn also supplies a plethora of utility functions for data preprocessing, model validation, and metric computations.

2.7.3 Gensim

Gensim [2] is a Python framework for vector space modeling. Gensim provides APIs for using word2vec and fastText, making it convenient for us to utilize a common platform to load both types of word embedding models and incorporate them into our system.

2.7.4 TensorFlow

TensorFlow [7] is "an open-source software library for numerical computation using data flow graphs", released by Google in 2015 in efforts to promote research in deep learning. Although not limited to neural networks, TensorFlow programs utilize multidimensional array data structures called *tensors* which serve as edges in a graph, connecting the nodes within a network. In other words, tensors hold the data that *flow* in and out of neurons, passing through layers in a neural network. This thesis work was conducted using TensorFlow 1.3.

GPU Offloading. TensorFlow is a computationally heavy framework that supports both CPU and GPU device types. As GPUs are built to handle mathematical operations much more efficiently than CPUs, offloading a TensorFlow program onto the GPU can drastically reducing training time; anecdotally, the time it took to run our

experiments reduced by nearly 20-fold. Despite the small size of our datasets, the GPU offloading made running numerous experiments with 10-fold cross-validation much less painful.

2.8 Datasets

The two datasets we research were provided to us by the *25th IEEE International Requirements Engineering Conference* (RE '17) call for data track papers[1]. Our work addresses the data track challenge area of requirements identification.

2.8.1 Security Requirements (SecReq)

Security is a category of non-functional requirement that embodies product, business, and customer safety — focusing on values such as ensuring system impenetrability, protecting business assets, and preserving user privacy. Despite the high stakes, designing and building secure systems is challenging due to the scarcity of software security expertise [25]. The acknowledgment of such challenges consequently launched the research and development for tactics to help non-security experts in identifying system areas that can introduce security vulnerabilities.

Table 2.3: SecReq dataset, broken down by SRS [25].

SRS	*# Requirements*	*# Security-Related*	*% Security-Related*
ePurse	124	83	66.9%
CPN	210	41	19.5%
GPS	173	63	36.4%
Combined	507	187	36.9%

[1] http://re2017.org/pages/submission/data_papers/ (accessed January 2018)

Table 2.4: Examples of security-related and not security-related requirements from SecReq.

SRS	*Requirements Text*	*Security-Related?*
`ePurse`	"All load transactions are on-line transactions. Authorization of funds for load transactions must require a form of cardholder verification. The load device must support on-line encrypted PIN or off-line PIN verification"	Yes
	"A single currency cannot occupy more than one slot. The CEP card must not permit a slot to be assigned a currency if another slot in the CEP card has already been assigned to that currency."	No
`CPN`	"On indication received at the CNG of a resource allocation expiry the CNG shall delete all residual data associated with the invocation of the resource."	Yes
	"It shall be possible to configure the CNG (e.g. firmware downloading) according to the subscribed services. This operation may be performed when the CNG is connected to the network for the first time, for each new service subscription/modification, or for any technical management (e.g. security, patches, etc.)."	No
`GPS`	"The back-end systems (multiple back-end systems may exist for a single card), which communicate with the cards, perform the verifications, and manage the off-card key databases, also shall be trusted."	Yes
	"If an Application implicitly selectable on specific logical channel(s) of specific card I/O interface(s) is deleted, the Issuer Security Domain becomes the implicitly selectable Application on that logical channel(s) of that card I/O interface(s)."	No

Knauss et al. assembled the SecReq dataset in efforts to promote research in security requirements elicitation automation and enhance security awareness [25]. The SecReq dataset is composed of three industrial SRS documents: Common Electronic Purse (`ePurse`), Customer Premises Network (`CPN`), and Global Platform Specification (`GPS`) [25]. Each document is broken down into individual requirements, labeled as either security-related or not security-related. The composition of the SecReq dataset allows for a straightforward binary classification task. Table 2.3 outlines the SRS breakdown and Table 2.4 provides requirements examples from each document.

2.8.2 Quality Attributes (NFR)

Table 2.5: NFR dataset, broken down by project and requirements type [14].

		Project ID															
Requirement Type	*Label*	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	*Total*
Availability	`A`	1	1	1	0	2	1	0	5	1	1	1	1	1	1	1	**18**
Legal	`L`	0	0	0	3	3	0	1	3	0	0	0	0	0	0	0	**10**
Look-and-Feel	`LF`	1	2	0	1	3	2	0	6	0	7	2	2	4	3	2	**35**
Maintainability	`MN`	0	0	0	0	0	3	0	2	1	0	1	3	2	2	2	**16**
Operational	`O`	0	0	6	6	10	15	3	9	2	0	0	2	2	3	3	**61**
Performance	`PE`	2	3	1	2	4	1	2	17	4	4	1	5	0	1	1	**48**
Scalability	`SC`	0	1	3	0	3	4	0	4	0	0	0	1	2	0	0	**18**
Security	`SE`	1	3	6	6	7	5	2	15	0	1	3	3	2	2	2	**58**
Usability	`US`	3	5	4	4	5	13	0	10	0	2	2	3	6	4	1	**62**
Total NFRs		8	15	21	21	37	44	8	71	8	15	10	20	19	16	12	*326*
Functional	`F`	20	11	47	25	36	26	15	20	16	38	22	13	3	51	15	**358**
Total		28	26	68	47	73	70	23	91	24	53	32	33	22	67	127	*684*

The *Quality Attributes* (NFR) dataset [4], also known as the *PROMISE* corpus, is a compilation of requirements specifications for 15 software projects developed by MS students at DePaul University as a term project for a Requirements Engineering course [14]. The dataset consists of 326 non-functional requirements (NFRs) of nine types and 358 functional requirements (FRs). Table 2.5 tabulates the distribution of requirement types among the 15 projects, and Table 2.6 provides examples of each type of requirement.

The NFR dataset lends itself to three different types of classification tasks: (1) binary classification of `NF` versus `F` requirements, and (2) binary classification of a `NF` requirement type, and (3) multi-label classification of various `NF` requirement types.

Table 2.6: Examples of requirements of different types from NFR.

Label	*Requirements Text*
A	"The RFS system should be available 24/7 especially during the budgeting period. The RFS system shall be available 90% of the time all year and 98% during the budgeting period. 2% of the time the system will become available within 1 hour of the time that the situation is reported."
L	"The System shall meet all applicable accounting standards. The final version of the System must successfully pass independent audit performed by a certified auditor."
LF	"The website shall be attractive to all audiences. The website shall appear to be fun and the colors should be bright and vibrant."
MN	"Application updates shall occur between 3AM and 6 AM CST on Wednesday morning during the middle of the NFL season."
O	"The product must work with most database management systems (DBMS) on the market whether the DBMS is colocated with the product on the same machine or is located on a different machine on the computer network."
PE	"The search for the preferred repair facility shall take no longer than 8 seconds. The preferred repair facility is returned within 8 seconds."
SC	"The system shall be expected to manage the nursing program curriculum and class/ clinical scheduling for a minimum of 5 years."
SE	"The product shall ensure that it can only be accessed by authorized users. The product will be able to distinguish between authorized and unauthorized users in all access attempts."
US	"If projected the data must be readable. On a 10x10 projection screen 90% of viewers must be able to read Event / Activity data from a viewing distance of 30."
F	"System shall automatically update the main page of the website every Friday and show the 4 latest movies that have been added to the website."

2.9 Prior Work

2.9.1 Classifying Security Requirements

Naïve Bayes Approach

Knauss et al. made a first attempt at classifying security requirements within the SecReq dataset with the Naïve Bayes approach [26]. They conducted two sets of experiments using 10-fold cross-validation for each combination of training and test sets:

1. *Single Domain.* Train a classifier with one individual SRS and test on another individual SRS.
2. *Multi Domain.* Train a classifier with a combination of SRS documents and test on an individual SRS.

The experiments are designed to gauge the effectiveness of training a classifier to use in classifying security requirements in different domains, consequently assessing whether overfitting can be avoided.

The results are aggregated in Table 2.7. Knauss et al. deemed a classifier to be useful if it achieves at least 70% recall and 60% precision [26]. Unsurprisingly, all experiments with classifiers trained and tested within the same domain(s) pass the test, whereas almost all the experiments with classifiers tested on an foreign domain do not.

Table 2.7: SecReq Naïve Bayes classification experiments by Knauss et al. [26].

Experiment	*Training Set*	*Test Set*	*Recall*	*Precision*	*F_1-score*
Single Domain	ePurse	**ePurse**	**0.93**	**0.83**	**0.88**
		CPN	0.54	0.23	0.33
		GPS	0.85	0.43	0.57
	CPN	ePurse	0.33	0.99	0.47
		CPN	**0.95**	**0.98**	**0.96**
		GPS	0.19	0.29	0.23
	GPS	ePurse	0.48	0.72	0.58
		CPN	0.65	0.29	0.40
		GPS	**0.92**	**0.81**	**0.86**
Multi Domain	ePurse + CPN	ePurse	0.95	0.80	0.87
		CPN	0.85	1.00	0.92
		GPS	0.56	0.51	0.53
		ePurse + CPN	**0.93**	**0.81**	**0.87**
	ePurse + GPS	ePurse	0.98	0.78	0.87
		CPN	0.85	0.26	0.40
		GPS	0.85	0.80	0.82
		ePurse + GPS	**0.96**	**0.80**	**0.87**
	CPN + GPS	ePurse	0.31	0.84	0.46
		CPN	0.75	0.88	0.81
		GPS	0.88	0.81	0.84
		CPN + GPS	**0.87**	**0.82**	**0.85**
	ePurse + CPN + GPS	ePurse	0.95	0.80	0.87
		CPN	0.85	0.94	0.89
		GPS	0.88	0.78	0.83
		ePurse + CPN + GPS	**0.91**	**0.79**	**0.84**

2.9.2 Classifying Quality Attributes

Keyword Mining Approach

Cleland-Huang et al. first conducted a small experiment to evaluate how effective a simple keyword mining approach can be when classifying non-functional requirements [14]. The team mined security and performance catalogs to extract sets of keywords associated with security and performance. Requirements containing words from the security keyword set were predicted as security NFRs, and those containing words

from the performance keyword set were predicted as performance NFRs. Requirements containing words from both were likewise classified as both, and requirements containing none were classified as neither.

The security classifier scored 79.8% for recall and 56.7% for precision, whereas the performance classifier scored 60.9% for recall and 39.6% for precision. The poor precision is due to the fact that many of these keywords were shared by other types of NFRs. The researchers did not replicate this experiment with any other NFR types due to the difficulty in finding catalogs related to the quality attribute to mine keywords from.

Weighted Indicator Approach

Cleland-Huang et al. next proposed a weighted indicator approach based on information retrieval to detect and classify NFRs [14]. The classifier is built on an explicit supervised learning-like model with pre-labeled training sets and a manual feature extraction process.

The method is detailed as follows: The requirements are first preprocessed with stop word removal and stemming. Using a pre-labeled training set, *indicator terms* for each NFR type are then mined and assigned probabilistic weights. Afterward, using the indicator terms, a requirement can be classified as a certain NFR type if its computed probability score beats a chosen threshold.

The researchers ran 15 iterations of the *leave-one-out* cross-validation technique, partitioning 14 out of the 15 projects to use as the training set and reserving the last project for validation. Table 2.8 showcases the results from selecting the 15 terms with the highest weights as indicator terms and choosing a threshold value of 0.04.

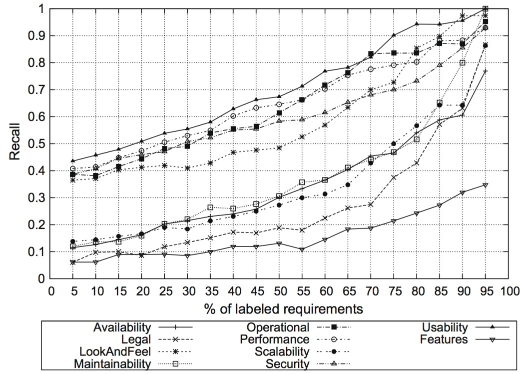
Table 2.8: Results from using top-15 terms at classification with threshold value of 0.04, by Cleland-Huang et al. [14].

NF *Type*	*Label*	*Recall*	*Precision*
Availability	A	0.8889	0.1111
Legal	L	0.7000	0.1628
Look and Feel	LF	0.5143	0.1169
Maintainability	MN	0.8824	0.1087
Operational	O	0.7213	0.1137
Performance	PE	0.6250	0.2727
Scalability	SC	0.7222	0.1111
Security	SE	0.8070	0.1840
Usability	US	0.9839	0.1442
	Average	*0.7669*	*0.1416*

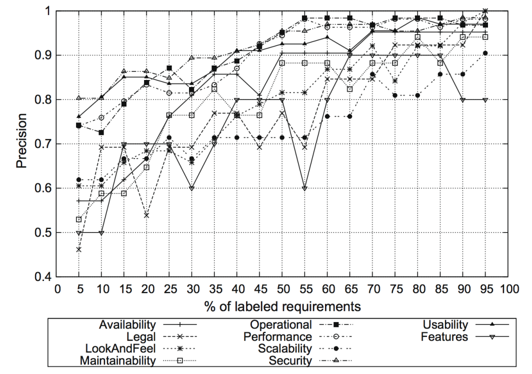
Semi-supervised Learning Approach

Fully-supervised learning requires a large amount of pre-labeled samples for training, meaning analysts need to manually review the requirements and make decisions as to which category the requirement belongs to. In hopes to alleviate such a time consuming prerequisite, Casamayor et al. [11] proposed a semi-supervised learning approach to NFR classification. The approach requires a reduced amount of pre-labeled data by incorporating unlabeled data into the learning process through a semi-supervised algorithm called *Expectation Maximization* (EM), which they built with Bayesian classifiers. They employed the *one-vs.-all* strategy for multi-label classification, where a binary classifier is trained for each class and during validation, requirements are classified as the class from the binary classifier that produces the highest score.

The requirements documents first undergo normalization, stop word removal, and stemming before being transformed into TF-IDF vectors. Experiments were run using stratified 10-fold cross-validation.



(a) Recall



(b) Precision

Figure 2.6: Recall and precision results from one-vs.-all multi-label classification of NFR types from Casomayor et al. [11].

.

Casomayor et al. demonstrate that their semi-supervised approach beats the performance of supervised methods like Naïve Bayes, k-nearest neighbors, and TF-IDF. Figure 2.6 showcases the recall and precision scores for one-vs.-all multi-label classification of NFR types over a range of volumes of pre-labeled data.

Chapter 3

DESIGN AND IMPLEMENTATION

To restate our research questions at hand, our goal for this thesis is to evaluate the effect of two deep learning methodologies applied directly to the domain of software requirements documents and the task of requirements classification. Specifically, we aim to investigate the following:

1. **RQ1:** The feasibility of deep learning models, namely CNNs, on requirements document analysis.
2. **RQ2:** The efficacy of pre-trained word embeddings in requirements document vectorization.

The unique properties of software requirements documents pose a number of concerns when considering deep learning techniques as the nature of the data does not align with the conditions conventionally well-suited for deep learning applications. Specifically, the following attributes of our data pique our interest:

1. Software requirements documents tend to be short in length, resulting in feature-poor data samples. Deep learning analyses typically require feature-rich data, so in the case of text, translates to much lengthier documents.
2. Software requirements datasets tend to be shallow in volume, in the order of hundreds of samples. Deep learning is typically employed to analyze massive volumes of data, in the order of millions or billions.

We want to evaluate whether the usage of feature-rich word representations provided through word embeddings can enrich the features of these documents and make

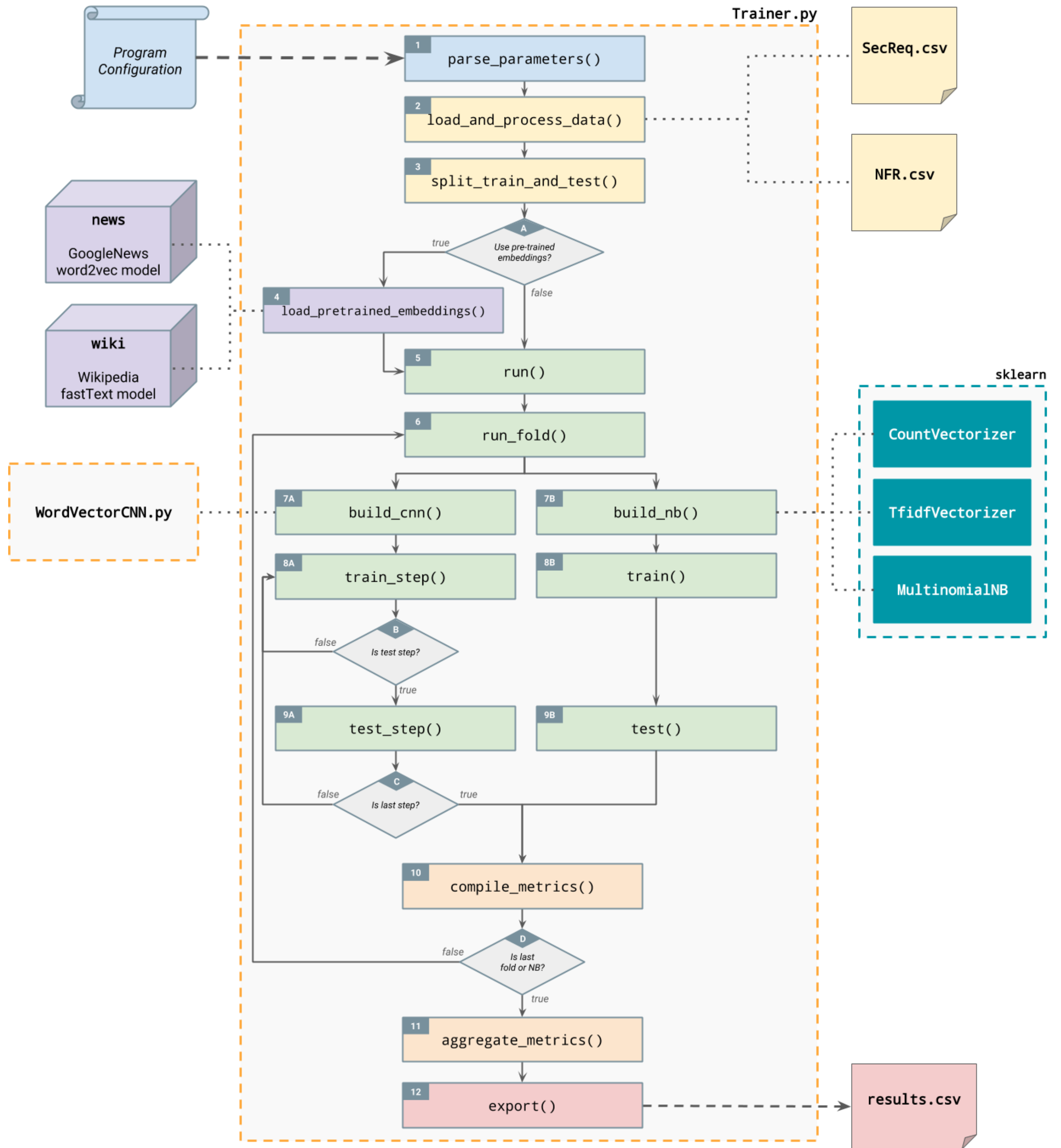


Figure 3.1: System design activity diagram.

up for their sparseness. Especially since requirements documents are often characterized by formal writing and domain-specific jargon, the words that make up the requirement text must be key in identifying characteristics of individual requirements. Since word embeddings are claimed to preserve semantic relationships between words that are otherwise lost in traditional vectorization methods such as TF-IDF [34, 35], we want to investigate whether these advantages can counteract the feature-poor quality of requirements samples.

In order to study these inquiries, we needed to design an instrument that can orchestrate the data preparation, classifier training, and performance evaluation necessary to assess the impact of the technologies mentioned above against our concerns with the nature of our domain. We built such a system with the help of machine learning tools such as TensorFlow, Scikit-Learn, Gensim, and NLTK. Figure 3.1 showcases an activity diagram that breaks down the operation flow within our system. The system features one flow to train a Naïve Bayes classifier using Scikit-Learn's `MultinomialNB`[1] implementation, as well as a flow to train a one-layer text classification CNN modeled after the sentence classification CNN design proposed by Kim et al. [23] and illustrated in Figure 2.4. The following sections walk through the steps of the activity diagram, further discussing each operation in detail.

3.1 Run Configurations

Our system accepts a set of command line configurations (or parameters) for each experiment run, categorized and detailed in Table 3.1. The program configurations are parsed in Block 1 of the activity diagram in Figure 3.1.

[1] `sklearn.naive_bayes.MultinomialNB`

Table 3.1: Run configuration options.

Category	*Configuration*	*Type*	*Description*
Dataset	Training Set	`str`	Dataset for training.
	Test Set	`str`	Dataset for testing.
	Label(s)	`set(str)`	Label(s) to classify.
Validation	Cross-Validation	`bool`	Whether to run k-fold cross-validation.
	# of Folds (k)	`int`	If k-fold cross-validation is set, specify k (e.g., 10).
	Test Percentage	`float`	If k-fold cross-validation is not set, specify % of dataset reserved for test (e.g., 0.25).
Preprocessing	Stratification	`bool`	Whether to stratify the training and test sets.
	Stop Word Removal	`bool`	Whether to remove stop words from the corpora.
	Lemmatization	`bool`	Whether to lemmatize the corpora.
Classifier	CNN	`bool`	Whether to train a CNN classifier.
	Embeddings	`str`	Word embedding initialization (`"random"`, `"w2v"`, or `"fasttext"`).
	Filter Sizes	`set(int)`	Set of filter sizes (e.g., $\{1, 2, 3\}$).
	# of Filters	`int`	Number of filters per filter size (e.g., 128).
	# of Epochs	`int`	Number of epochs to train for (e.g., 140).
	NB	`bool`	Whether to train a NB classifier.
	Vectorizer	`str`	Vectorization method (`"count"` or `"tfidf"`).
Program	Seed	`int`	Seed to control randomization (e.g., 42).

3.2 Data Loading and Preprocessing

Next in the pipeline, the data is loaded and preprocessed (according to the program configurations) and converted into a uniform data format for our classifiers to accept. This occurs in Block 2 of the activity diagram in Figure 3.1.

3.2.1 Raw Data Formats

As the SecReq and NFR datasets come in different formats, we discuss processing both dataset separately.

SecReq

The SecReq dataset is comprised of three different SRS documents in the form of three similarly formatted semicolon-delimited CSV files: `ePurse-selective.csv`,

```
The CNG shall support mechanisms to authenticate itself to the NGN for
connectivity purposes.;sec
```

(a) Security-related

```
The CND should be able to support bootstrap capabilities in order to
retrieve network configuration data to connect the NGN.;nonsec
```

(b) Not security-related

Figure 3.2: Examples of SecReq raw data format.

`CPN.csv`, and `GPS.csv`. Each data point in the collection features two attributes: the requirements text and the requirements class label (`sec` or `nonsec`). Figure 3.2a showcases an example of a security-related requirement, whereas Figure 3.2b showcases an example of a non-security-related requirement.

NFR

The NFR dataset comes in an ARFF file (typically used to load Weka[2] programs) `nfr.arff`, which conveniently is also a comma-separated document that, by stripping some extraneous metadata, we can convert to a standard CSV. Each data point in the file features three attributes: the project ID, the requirements text (wrapped in single quotes), and the requirements class label (documented in Table 2.5). Figure 3.3 showcases an example of a performance (`PE`) requirement from Project 4.

```
4,‘The Disputes application shall support 350 concurrent users without any
degradation of performance in the application.’,PE
```

Figure 3.3: Example of NFR raw data format.

[2] `https://weka.wikispaces.com/`

3.2.2 Classifier Input Format

Our classifiers expect the data to be in the form of a list of requirements text strings (`x_text`) and an accompanying list of one-hot vectors representing the associated requirements class (`y`). More specifically, we convert the class label for each data sample into a one-hot vector with each index representing a class, with the value of 1 indicating the active class. For example, for the two classes for SecReq, the vector `[0, 1]` represents the positive class `sec` and `[1, 0]` represents the negative class `nonsec`.

The CSV file is first parsed into a `DataFrame`[3] for Python convenience, with each row in the `DataFrame` representing a single requirements sample. Each requirements sample undergoes the following preprocessing procedures:

1. Clean the requirements text string by first tokenizing it into words and then stripping it of punctuation and contractions.
2. If stop word removal or lemmatization is enabled, then each token is processed with the respective operation.
3. For each requirements sample, the cleaned tokens are stitched back together into a space-separated string and appended to the list `x_text`.
4. The corresponding class label is converted into a one-hot vector and appended to the list `y`.

After processing all requirements samples in the dataset, `x_text` and `y` are primed and ready for the rest of the system.

[3] `pandas.DataFrame`

3.2.3 Training and Test Set Division

After preprocessing, the dataset needs to be split into training and test sets according to the program configurations. This occurs in Block 3 of the activity diagram in Figure 3.1.

If we are training and testing on the same dataset, then a percentage p of the dataset is reserved for testing and the remainder for training. Otherwise, if we are training and testing on different datasets, then p% of the first dataset is pulled for testing and $(100 - p)$% percent of the second dataset is used for training.

If k-fold cross-validation is set, then we employ Scikit-Learn's `KFold`[4] (or if stratification is set, `StratifiedKFold`[5]) to evenly divide our samples into k training and test folds. Otherwise, we shuffle the samples and use the test percentage p specified in the program configuration.

3.2.4 Pre-trained Embedding Model Loading

If a pre-trained word embedding model is selected, then the model needs to be loaded into the system from the external model file. This occurs in Block 4 of the activity diagram in Figure 3.1.

The two available pre-trained embedding models we consider are the Google News word2vec model (later referred to as `news`) and the Wikipedia fastText model (later referred to as `wiki`), introduced in Section 2.4.2. We use Gensim's `KeyedVectors`[6] implementation to load either model into a common lookup table format. Section 3.3.1 further elaborates on how the pre-trained embedding model is utilized in document vectorization.

[4] `sklearn.model_selection.KFold`
[5] `sklearn.model_selection.StratifiedKFold`
[6] `gensim.models.KeyedVectors`

3.3 Classifier Training

Once all the necessary data is prepared, the next step is to run the experiment by building, training, and testing either a CNN or Naïve Bayes classifier. This occurs in Blocks 5-9 of the activity diagram in Figure 3.1, where the flows diverge at Block 7 depending on the classifier selected.

3.3.1 CNN

We designed a custom class `WordEmbeddingCNN` using the TensorFlow GPU framework, closely modeling the sentence classification CNN architecture proposed by Kim et al. [23] and described in Section 2.3.3. The CNN classifier is built in Block 7A of the activity diagram in Figure 3.1. Figure 3.4 illustrates the structure of our CNN.

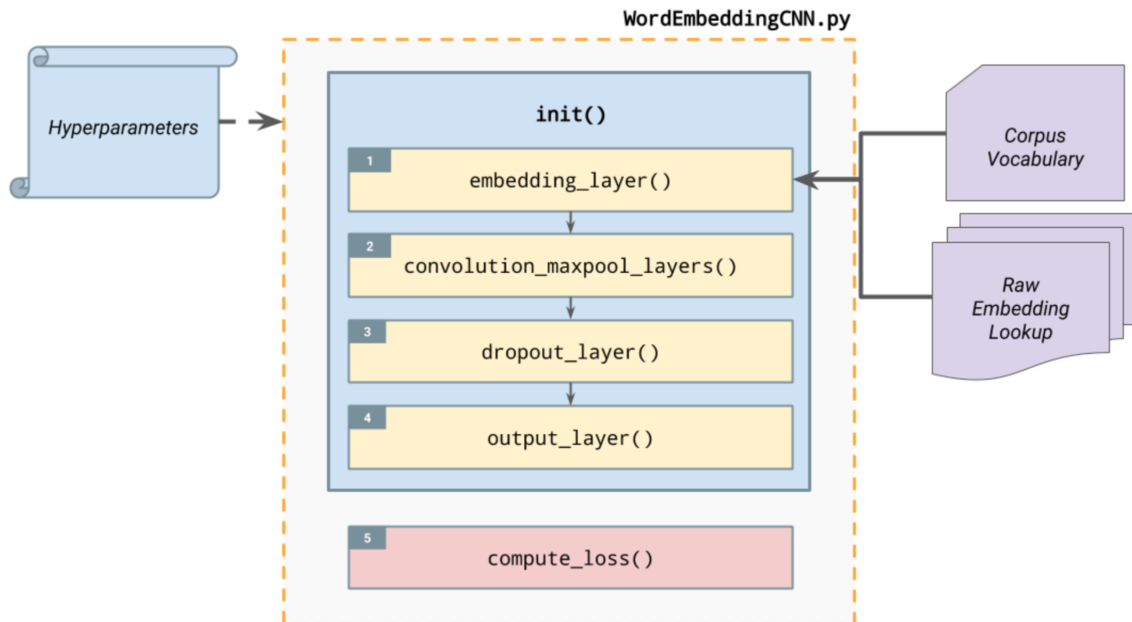


Figure 3.4: Illustration of `WordEmbeddingCNN` layer initialization.

Hyperparameters

Our `WordEmbeddingCNN` requires a set of hyperparameters to configure the CNN. Table 3.2 lists the parameters necessary to initialize the CNN.

Table 3.2: `WordEmbeddingCNN` hyperparameters.

Hyperparameter	*Variable*	*Description*
Sequence Length	n	Size of each input vector.
Number of Classes	$\|y\|$	# of available class labels.
Embedding Size	d	Size of the word embedding vectors.
Filter Sizes	H	Set of available filter sizes.
Number of Filters	m	# of filters per filter size.

Architecture

The architecture for our `WordEmbeddingCNN` closely resembles the one-layer sentence classification CNN structure introduced by Kim et al. [23], with the distinction that we use entire requirements as input rather than solely a single sentence.

Embedding Layer. The *embedding layer* is where the input tensor is transformed into an *embedding matrix*, meaning the raw input text is converted to a stack of their respective embeddings. This occurs in Block 1 of Figure 3.4.

In order to streamline the embedding matrix construction for this corpus, our first prerequisite is to create a lean vocabulary-to-embedding lookup table by matching the corpus vocabulary with its corresponding embedding from the raw embedding lookup. The product of this inner join can be referred to as the *corpus embedding lookup*. We serialize the corpus embedding lookup to a file for future runs with this corpus and embedding model combination. If a pre-trained model is not provided

(i.e., the `random` embedding configuration was selected), the values for the input embedding matrix are initialized from a uniform distribution.

To actually build the embedding matrix that will represent the input text, we (1) iterate through each token in the text, (2) search for its d-dimensional word embedding from the corpus embedding lookup, and (3) stack them together to create a $N \times d$ matrix, where N is the length of the longest requirement in the dataset. Words not present in the corpus embedding lookup are initialized as zero vectors of size d. Furthermore, for documents that contain fewer than N tokens, we pad the remainder of the embedding matrix with zero vectors. Table 3.3 illustrates an example embedding matrix where the embeddings are of size $d = 5$, the input text is of length $n = 7$, and the length of the longest document is $N = 10$. The term `"CEP"` is not recognized in the corpus embedding lookup, and thus is represented as a zero vector.

Table 3.3: Embedding matrix example ($d = 5, n = 7, N = 10$).

`the`	2.2435e−02	−1.2019e−02	3.6679e−02	−3.1872e−02	9.8377e−03
`CEP`	0	0	0	0	0
`card`	−7.4428e−02	−7.1579e−02	−1.3532e−01	3.4365e−02	−4.9633e−03
`must`	−1.4689e−03	−4.7007e−02	−5.4129e−02	−4.2301e−02	−7.2291e−02
`authenticate`	−1.5580e−02	3.1999e−02	3.4009e−02	6.7306e−02	2.1990e−02
`the`	2.2435e−02	−1.2019e−02	3.6679e−02	−3.1872e−02	9.8377e−03
`terminal`	−7.9057e−02	−1.1039e−01	−2.0125e−02	1.4814e−01	7.6921e−02
	0	0	0	0	0
	0	0	0	0	0
	0	0	0	0	0

Convolutional and Pooling Layers. Following the preparation in the embedding layer is a collection of *convolutional and pooling layers* acting in parallel, initialized in Block 2 of the activity diagram in Figure 3.4. Recall that a convolution represents an n-gram in the text sample and results in one feature. The CNN design by Kim et al.

calls for a single convolutional layer supporting multiple filter sizes, but TensorFlow requires each convolutional layer to support just one filter size. We can emulate the one-layer design by initializing a convolutional and max-pooling layer for each filter size h and keeping the layers parallel. Each convolutional layer is max-pooled and the output from all pooling is concatenated into a final feature vector to funnel to the dropout layer. Revisit Figure 2.4 for an illustration of how convolutions are built and pooled.

Dropout Layer. Specifically to our design, we follow the convolutional and pooling layers with a *dropout layer*, as shown in Block 3 of the activity diagram in Figure 3.4. We use the TensorFlow dropout API to facilitate the dropout of neurons given a dropout probability p. For our experiments, we kept $p = 0.5$ but other values can be explored in the future.

Output Layer. The *output layer*, initialized in Block 4 of the activity diagram in Figure 3.4, is where a softmax regularization function is applied to convert the output values from the dropout layer to normalized scores for class prediction.

Training

After building an instance of our `WordEmbeddingCNN`, we proceed to define the training and testing procedures for the given classification task.

Optimization Strategy. We need to declare an optimizer to optimize the performance of gradient descent, as explained in Section 2.3.2. In our design, we chose to employ TensorFlow's `AdamOptimizer`[7] implementation to help minimize the cross-entropy loss for each round of training.

[7] `tensorflow.train.AdamOptimizer`

Batch Iteration. The input points are divided into batches of a pre-configured size. As explained in Section 2.3.2, for each learning step s, a batch of inputs is propagated through the network, revising the weights of each neuron through forward and back propagation. In our design, we arbitrarily chose a batch size of 64 samples.

3.3.2 Naïve Bayes

For the Naïve Bayes classifier, we employ Scikit-Learn's `MultinomialNB` implementation. The NB classifier is built in Block 7B of the activity diagram in Figure 3.1.

Feature Extraction

The features that we supply our `MultinomialNB` classifier is a vectorized version of our input text. Rather than word embeddings, we solely use the count and TF-IDF vectorization methods discussed in Section 2.4.2 using Scikit-Learn's `CountVectorizer`[8] and `TfidfVectorizer`[9] implementations.

Training

Training the NB classifier is straightforward compared to the CNN. We simply fit the training data to the classifier using `MultinomialNB`'s built-in functions.

3.3.3 Testing and Result Compilation

For each fold (or the entire run if k-fold cross-validation is not set), the trained classifier is tested against its designated test set. The performance metrics discussed in Section 2.6 are compiled for each test (Block 10) and averaged at the end of the entire run (Block 11).

[8] `sklearn.feature_extraction.text.CountVectorizer`
[9] `sklearn.feature_extraction.text.TfidfVectorizer`

3.4 Exportation

After the final metrics are compiled into a Pandas `DataFrame`, the results are exported to a CSV file for records (Block 12).

Formatting Scripts

As each individual run is exported to a single CSV file, comparing the metrics from various runs can be cumbersome with the sheer volume of result files. We composed various Python scripts to help aggregate the loose data into more cohesive collections of results.

Chapter 4

EXPERIMENTS AND RESULTS

Recall that our research questions for this thesis are to assess the feasibility of CNNs applied to the domain of software requirements (RQ1), and measure the influence of pre-trained word embeddings in vectorizing our software requirements classification (RQ2). In order to accomplish this, we define various classification problems from our two datasets. For each classification problem, we train three types of classifiers:

1. *Naïve Bayes*, to serve as a baseline.
2. *CNN with random word embeddings*, to assess the feasibility of employing CNNs on this task (RQ1).
3. *CNN with pre-trained word embeddings*, to assess the influence of pre-trained word embeddings on requirements vectorization (RQ2).

The SecReq and NFR datasets lend themselves to very straightforward classification tasks. Table 4.1 outlines the primary classification problems we consider. We include NFR-SE as a primary classification problem because of our interest in security requirements from SecReq.

Table 4.1: SecReq and NFR primary classification problems.

Name	*Dataset*	*Classification Type*	*Description*
SecReq-SE	SecReq	Binary	Security-related (SE) vs. not security-related
NFR-NF	NFR	Binary	Non-functional (NF) vs. functional (F)
NFR-Types	NFR	Multi	A vs. L vs. LF vs. MN vs. O vs. PE vs. SE vs. SC vs. US
NFR-SE	NFR	Binary	Security-related (SE) vs. not security-related NFRs

Using the system described in Chapter 3, we designed a suite of experiments targeting our research questions, incorporating the primary classification problems defined in Table 4.1. For each run of our system, we (1) seed the random state to control randomness, allowing for reproducibility, and (2) stratify the k-folds in cross-validation to promote fair representation of the dataset in each fold.

The following list provides an overview of the experiments conducted:

1. *Experiment 1: Naïve Bayes Baselines.* Reproduce or establish Naïve Bayes baselines for the primary classification problems (refer to Table 4.1).

2. *Experiment 2: Optimal CNN Models for Binary Classification.* Determine the optimal CNN model for each word embedding configuration for the binary classification problems.

3. *Experiment 3: Optimal CNN Models for Multi-label Classification.* Determine the optimal CNN model for each word embedding configuration for the NFR type multi-label classification problem. Compare the performance of a single multi-label classifier versus the performance of individual binary classifiers for each NFR type.

4. *Experiment 4: Epoch Convergence.* Using the optimal CNN models determined from Experiment 2, evaluate where the CNN performance converges from the number of training epochs.

5. *Experiment 5: Cross-Dataset Security Requirements Classification.* Evaluate the potential of overfitting and the quality of the models by assessing the performance of security classifiers trained on one dataset and validated on another.

Initial explorations of Experiments 1-2 on the SecReq-`SE` and NFR-`NF` problems are reported in our paper published to the RE '17 conference [17]. The discoveries from our RE paper helped shape the design of the experiments explored in this thesis.

4.1 Experiment 1: Naïve Bayes Baselines

In Experiment 1, we attempt to either establish (or reproduce, in the case of SecReq-`SE`) a baseline to refer to in our later experiments that involve deep learning. We choose Naïve Bayes as it is a straightforward text classification method often employed for baseline metrics. As explained in our system design, we utilize Scikit-Learn's `MultinomialNB` as well as their `CountVectorizer` and `TfidfVectorizer` feature extractors. We run our Naïve Bayes classifiers with 10-fold cross-validation.

The following subsections showcase the results from using `CountVectorizer`, as the results from word frequency produce superior results to TF-IDF. Refer to Appendix A for the TF-IDF results.

4.1.1 SecReq-`SE`

For SecReq-`SE`, we attempt to train a security requirements classifier to reproduce the Naïve Bayes results from Knauss et al. [26]. As discussed in Section 2.8.1, Knauss et al. built their own Naïve Bayes classifier using word presence as features and trained it on the SecReq dataset [26].

Table 4.2: Naïve Bayes results for SecReq-`SE` binary classification.

Method	*Recall*	*Precision*	*F_1-score*
`MultinomialNB + CountVectorizer`	0.888	0.791	0.834
Knauss et al. [26]	*0.91*	*0.78*	*0.84*

Table 4.2 compares our results with the original metrics from Knauss et al. Our metrics run fairly close with minor difference likely attributed to the disparity in classifier implementation and feature extraction (Knauss et al. used word presence whereas we use word frequency).

4.1.2 NFR-NF

For NFR-NF, we establish a baseline for identifying non-functional requirements versus functional requirements. Training a binary NF vs. F Naïve Bayes classifier on the NFR corpus produces strong baseline performance with a 92.1% F_1-score for identifying NF requirements. The metrics for each class label are broken down in Table 4.3.

Table 4.3: Naïve Bayes (with CountVectorizer) results for NFR-NF binary classification.

Requirement Type	*Recall*	*Precision*	*F_1-score*
NF	**0.926**	**0.915**	**0.921**
F	0.874	0.895	0.884
Average	*0.900*	*0.905*	*0.903*

4.1.3 NFR-Types

For NFR-Types, we want to establish baseline metrics for distinguishing all nine NF types in the dataset. To mimick previous studies in classifying NFR types [14], we first remove all functional (F) requirements from the dataset, leaving 326 non-functional requirements. Because this task involves the classification of multiple labels, we evaluate two approaches: (1) training a single multi-label classifier, and (2) training multiple binary classifiers, one for each label. The metrics for both classification tasks are broken down in Table 4.4.

Training classifiers against the NF types on the NFR corpus produces more variable metrics, with some labels performing much better than others. This is mainly due to the sheer difference in volume between the class label samples, visualized in Table 2.5. For example, our classifier fails to identify a single legal (L) requirement, likely because legal samples make up only 3% of the dataset. Likewise, classes with substantial presence in the dataset such as operational (O), performance (PE), security (SE), and

Table 4.4: Naïve Bayes (with `CountVectorizer`) results for NFR-Types multi-label classification and individual NFR label binary classification.

Classification Type	*Metric*	A	L	LF	MN	O	PE	SC	SE	US	*Average*	*Total*
Multi	*Rec.*	0.250	0.0	0.525	0.0	0.805	0.780	0.300	0.862	0.893	0.491	-
	Prec.	0.367	0.0	0.867	0.0	0.603	0.778	0.500	0.825	0.587	0.503	-
	F_1	0.297	0.0	0.654	0.0	0.690	0.779	0.375	0.843	0.708	0.497	-
	TP	*4.5*	*0.0*	*18.4*	*0.0*	*49.1*	*37.4*	*5.4*	*50.0*	*55.4*	-	*220.2*
Binary	*Rec.*	0.150	0.0	0.317	0.100	0.288	0.663	0.200	0.638	0.574	0.326	-
	Prec.	0.300	0.0	0.800	0.100	0.700	0.922	0.300	0.958	0.885	0.552	-
	F_1	0.200	0.0	0.454	0.100	0.408	0.771	0.240	0.766	0.696	0.410	-
	TP	*2.7*	*0.0*	*11.1*	*1.6*	*17.6*	*31.8*	*3.6*	*37.0*	*35.6*	-	*141.0*

usability (`US`) produce more promising results.

One-vs.-All. With the second approach of training a binary classifier for each `NF` type, ideally we would set up a *one-vs.-all* classification experience, where we train a binary classifier for each individual label, and during classification, assign the label to a sample from the binary classifier that earned the highest score. Unfortunately, as addressed in our future work, our system design does not yet support this infrastructure, leaving us unable to facilitate a proper comparison between multi-label classifiers and one-vs.-all classification.

4.1.4 NFR-`SE`

The performance for NFR-`SE` is included in Table 4.4 under `SE` binary classification. The Naïve Bayes classifier achieves a recall of 63.8% and very high precision of 95.8%.

4.2 Experiment 2: Optimal CNN Models for Binary Classification

Experiment 2 is an extension to the work from our RE Data Challenge paper [17] and is the core of this thesis. In our RE paper, we conducted an initial investigation of word2vec and CNN configurations on SecReq-`SE` and NFR-`NF`, comparing perfor-

mances of CNNs of two different filter counts (30 and 50) trained over a range of epochs (20 to 100), both with and without the incorporation of pre-trained word2vec word embeddings [17]. The results imply that these classifiers do grant a lift in performance in comparison to Naïve Bayes.

We can refer to our work from the RE paper as our *pilot* experiment — a trial assessing whether we should allocate more time and resources to research the potential of these methods. In the pilot experiment, we set the filter sizes of our CNN to be $\{3, 4, 5\}$ (as suggested by Kim et al. [23]) without full comprehension of what filter sizes represent in respect to our corpus. Now that we understand that a filter of size n harbors n-grams of our text, we deduce that 4 and 5-grams are much too large for short documents like software requirements. Realizing this, we design Experiment 2 as an exhaustive search for the optimal CNN model for each word embedding method (`random`, `word2vec`, `fastText`) over a range of filter sizes (subsets of $\{1, 2, 3\}$) and number of filters per size (16, 32, 64, 128, 256).

We train each CNN for 140 epochs and validate through 10-fold cross-validation. In the pilot experiment, we have the number of epochs as an independent variable; for this experiment, we select a number seemingly high enough to reach performance convergence but not too high as to add unnecessary training time. Our selection for number of epochs is later validated in Experiment 4, discussed in Section 4.4.

We run this experiment on all the three primary binary classification problems and report the highest scoring CNN model for each word embedding type, resulting in three CNN models for each classification problem. We also compare the performance of the three optimal CNN models with the other word embedding initializations.

Scoring Function

In order to determine the optimal CNN model, we need to devise a scoring function S to measure how good a CNN model is. We consider the following criteria for the scoring function:

- F_1*-score* (f), to measure the recall and precision of the model applied on a given dataset.
- *Loss* (ℓ), to measure how firm the model is with its predictions.
- *Filter Sizes* (H) and *# of Filters* (m), to assess the complexity of the shape of the model.

The scoring function, shown in Equation 4.1, favors higher F_1-scores, lower loss scores, lower CNN complexity.

$$S(f, \ell, H, m) = \frac{1}{2} \cdot \left(\frac{f}{\max_{f_1 \dots f_T}} + \frac{\min_{\ell_1 \dots \ell_T}}{\ell} \right) + C(H, m) \tag{4.1}$$

Let T be the total number of configurations considered. The final score S is equal to the average between the rank of F_1-score f and loss ℓ relative to the highest F_1-score and lowest loss among the collection of configurations, plus a penalty C.

$$C(H, m) = -0.01 \cdot \left(|H| \cdot \frac{m}{\max_M} \right) \tag{4.2}$$

We define the penalty C in Equation 4.2 as the negative product between $|H|$, the number of filter sizes (e.g., if $H = \{1, 2, 3\}$, then $|H| = 3$), and m, the number of filters per size normalized by the maximum filter count considered. The product is reduced by two decimal places to fit the range of S.

4.2.1 SecReq-SE

Table 4.5 reports an optimal CNN model for each word embedding type and its performance on SecReq-SE. Table 4.6 showcases the loss and F_1-scores of each optimal CNN model identified in Table 4.5 applied to the other word embedding initializations.

Table 4.5: SecReq-SE: Optimal CNN model results, trained with 140 epochs.

Embedding Type	*Filter Sizes*	*# Filters per Size*	*Loss*	*Accuracy*	*Recall*	*Precision*	*F_1-score*	*S*
`random`	$\{1, 2\}$	16	0.273	0.894	0.815	0.880	0.846	*0.755*
`word2vec`	$\{1, 2\}$	128	0.163	0.928	0.877	**0.919**	0.897	*0.971*
`fastText`	$\{1, 2, 3\}$	32	**0.161**	**0.936**	**0.915**	0.907	**0.911**	***0.993***

Table 4.6: SecReq-SE: Loss and F_1-score from optimal CNN models applied to each word embedding type.

Filter Sizes × # of Filters	`random`		`word2vec`		`fastText`	
	Loss	*F_1*	*Loss*	*F_1*	*Loss*	*F_1*
$\{1, 2\} \times 16$	0.273	0.846	0.180	0.896	0.190	0.884
$\{1, 2\} \times 128$	0.348	0.856	0.163	0.897	0.168	0.898
$\{1, 2, 3\} \times 32$	0.316	0.853	0.184	0.897	**0.161**	**0.911**

Recall from Table 4.2 in Experiment 1 the Naïve Bayes baseline metrics for SecReq-SE: 88.8% recall and 79.1% precision from our implementation and 91% recall and 78% precision from Knauss et al. With the plain CNN without pre-trained word embeddings, we already achieve a significant 9-10% boost in precision but with the expense of recall. With the incorporation of pre-trained word embeddings to our CNNs, we can achieve an even greater boost of about 12-14% precision from the baseline while preserving recall. Both models with `word2vec` and `fastText` perform comparably, but `fastText` achieves the closest recall to the baseline.

4.2.2 NFR-`NF`

Table 4.7 reports an optimal CNN model for each word embedding type and its performance on NFR-`NF`. Table 4.8 showcases the loss and F_1-scores of each optimal CNN model identified in Table 4.7 applied to the other word embedding initializations.

Table 4.7: NFR-`NF`: Optimal CNN model results, trained with 140 epochs.

Embedding Type	*Filter Sizes*	*# Filters per Size*	*Loss*	*Accuracy*	*Recall*	*Precision*	*F_1-score*	*S*
`random`	$\{1, 2, 3\}$	32	0.287	0.923	0.959	0.918	0.938	*0.862*
`word2vec`	$\{1, 2\}$	64	**0.212**	0.923	0.943	**0.929**	0.936	***0.990***
`fastText`	$\{2, 3\}$	32	0.223	**0.932**	**0.962**	**0.929**	**0.945**	*0.974*

Table 4.8: NFR-`NF`: Loss and F_1-score from optimal CNN models applied to each word embedding type.

Filter Sizes × # of Filters	`random`		`word2vec`		`fastText`	
	Loss	F_1	*Loss*	F_1	*Loss*	F_1
$\{1, 2, 3\} \times 32$	0.287	0.938	0.219	0.942	0.229	**0.945**
$\{1, 2\} \times 64$	0.308	0.928	**0.212**	0.936	0.226	0.942
$\{2, 3\} \times 32$	0.318	0.932	0.222	0.939	0.223	**0.945**

Recall from Table 4.3 in Experiment 1 that Naïve Bayes for NFR-`NF` yields a baseline of 92.6% recall and 91.5% precision. Both CNN models equipped with pre-trained embeddings safely beat the baseline performance by a few percentage points, with `fastText` leading recall improvement by 3.6% and precision by 1.4%, an overall lift of 2.4% in F_1-score. The optimal CNN model without pre-trained embedding support produces comparable results to the baseline.

4.2.3 NFR-`SE`

Table 4.9 reports an optimal CNN model for each word embedding type and its performance on NFR-`SE`. Table 4.10 showcases the loss and F_1-scores of each optimal

CNN model identified in Table 4.9 applied to the other word embedding initializations.

Table 4.9: NFR-`SE`: Optimal CNN model results, trained with 140 epochs.

Embedding Type	*Filter Sizes*	*# Filters per Size*	*Loss*	*Accuracy*	*Recall*	*Precision*	*F_1-score*	*S*
`random`	$\{1\}$	128	0.319	0.918	0.593	**0.957**	0.732	*0.632*
`word2vec`	$\{1,2\}$	32	0.148	0.940	0.743	0.929	0.826	*0.930*
`fastText`	$\{1\}$	64	**0.134**	**0.951**	**0.774**	0.953	**0.854**	***0.991***

Table 4.10: NFR-`SE`: Loss and F_1-score from optimal CNN models applied to each word embedding type.

Filter Sizes × # of Filters	`random`		`word2vec`		`fastText`	
	Loss	*F_1*	*Loss*	*F_1*	*Loss*	*F_1*
$\{1\} \times 128$	0.319	0.732	0.161	0.851	0.139	0.847
$\{1,2\} \times 32$	0.370	0.685	0.148	0.826	0.153	0.838
$\{1\} \times 64$	0.302	0.697	0.156	0.850	**0.134**	**0.854**

Recall from Table 4.4 in Experiment 1 that Naïve Bayes for NFR-`SE` yields a baseline of 63.8% recall and 95.8% precision. With such a high baseline, none of the CNN configurations beat the baseline precision. The `fastText`-powered CNN configuration offers the highest recall, providing a boost of 13.6%, while performing at baseline precision. Curiously, `random` performs at baseline precision but does not meet baseline recall, whereas `word2vec` beats baseline recall but fails to reach baseline precision.

4.2.4 Discussion

For Experiment 2, we compose a scoring function and run an exhaustive search to find the optimal CNN model for each word embedding method for each classification task. Our general conclusions mirror those from our pilot experiment — CNNs can potentially provide a lift in performance metrics within the SecReq and NFR datasets,

especially when employing pre-trained word embedding models to our training.

In this experiment, we deviate from the pilot by (1) keeping the number of training epochs for constant, and (2) running all the combinations of $\{1, 2, 3\}$ filter sizes. In addition, we introduce a new pre-trained word embedding model: `fastText` trained on the Wikipedia corpus. Judging based on our results for each classification problem, it can be observed that both `word2vec` and `fastText` pre-trained word embedding models generally offer an improvement in performance, with `fastText` in the lead.

4.3 Experiment 3: Optimal CNN Models for Multi-label Classification

Experiment 3 is a direct continuation of Experiment 2, carrying on its motivations and procedures but applied to a different classification task. While we group all the binary problems together in Experiment 2, we devote Experiment 3 to the multi-label problem, namely the classification of `NF` types. As discussed in Section 4.1.3, we do not yet have the support for one-vs.-all classification, so we cannot provide a proper comparison between the performance of a single multi-label classifier versus one-vs.-all classification.

In this section we orchestrate an exhaustive search for the optimal CNN configurations for each word embedding initialization for the multi-label CNN classifier for NFR-Types. We employ the same scoring function introduced in Experiment 2, with the adjustment of f being the average F_1-score between all the labels. Afterward, we train CNNs for every class label with each of the optimal models. Although the results from the multi-label classifier and the individual binary classifiers cannot be compared at face value, we can still make some observations about the nature of the problem at hand.

4.3.1 NFR-Types: Multi-label

Table 4.11 organizes the results for the optimal CNN model for each word embedding initialization for multi-label classification of NFR types.

Table 4.11: NFR-Types: Optimal multi-label CNN model results, trained with 140 epochs.

Embedding Type	*Filter Sizes*	*# Filters per Size*	*Loss*	*Acc.*	*Metric*	A	L	LF	MN	O	PE	SC	SE	US	*Avg.*	*Total*	*S*
random	{1}	256	0.909	0.731	*Rec.*	0.700	0.550	0.725	0.200	0.750	0.813	0.533	0.848	0.798	0.657	-	*0.751*
					Prec.	0.683	0.600	0.777	0.200	0.707	0.762	0.600	0.843	0.732	0.656	-	
					F_1	0.692	0.573	0.750	0.200	0.728	0.787	0.565	0.845	0.764	0.655	-	
					TP	*12.3*	*6.0*	*27.2*	*3.2*	*43.1*	*36.6*	*10.8*	*48.9*	*45.4*	-	*233.4*	
word2vec	{1, 2}	256	**0.611**	**0.804**	*Rec.*	0.750	0.700	0.733	0.550	0.883	0.797	0.567	0.893	0.876	0.750	-	***0.977***
					Prec.	0.867	0.800	0.813	0.517	0.745	0.849	0.850	0.855	0.854	0.794	-	
					F_1	0.804	0.747	0.771	0.533	0.808	0.822	0.680	0.873	0.865	**0.767**	-	
					TP	*15.6*	*8.0*	*28.5*	*8.3*	*45.4*	*40.7*	*15.3*	*49.6*	*52.9*	-	***264.3***	
fastText	{1, 2, 3}	256	0.632	0.792	*Rec.*	0.750	0.700	0.817	0.400	0.819	0.797	0.550	0.907	0.845	0.732	-	*0.947*
					Prec.	0.800	0.767	0.855	0.500	0.780	0.823	0.700	0.817	0.789	0.759	-	
					F_1	0.832	0.674	0.847	0.450	0.800	0.803	0.663	0.888	0.859	0.757	-	
					TP	*15.6*	*7.0*	*29.8*	*7.2*	*45.2*	*39.7*	*14.4*	*49.7*	*53.3*	-	*262.0*	

Recall in Experiment 1, we establish a Naïve Bayes baseline for the multi-label classifier shown in Table 4.4. The results from Table 4.11 collectively produce a generous boost in overall performance, from a 15.8% F_1-score and 13.2 TP increase with the `random` configuration to a 25-27% F_1-score and 41.8-44.1 TP increase with `fastText` and `word2vec`. As for each individual class label, the multi-label CNNs generally offer a modest improvement to the metrics for the well-represented classes (i.e., `O`, `PE`, `SE`, `US`).

However, the impact of CNNs and word embeddings can best be observed through the change in metrics for classifying under-represented class labels in the data (i.e., `A`, `L`, `MN`). While the multi-label Naïve Bayes classifier produces a F_1-score of 29.7% and 4.5 TP with classifying `A` requirements, the multi-label CNNs provide a 39.5-53.5% raise in F_1-score and 7.8-11.1 additional TP, from 66% with `random` to 78% with `word2vec`. In addition, while the multi-label Naïve Bayes classifier could not identify a single `L` and `MN` requirement, the multi-label CNNs identify as many as 8 true positives for either label.

4.3.2 NFR-Types: Binary CNNs

Table 4.12 organizes the results from configuring binary CNNs for classifying each NFR type with the optimal models discovered above in Section 4.3.1.

Table 4.12: NFR-Types: Individual binary CNN results, trained with 140 epochs.

Embedding Type	*Filter Sizes*	*# Filters per Size*	*Metric*	A	L	LF	MN	O	PE	SC	SE	US	*Avg.*	*Total*
random	{1}	256	*Rec.*	0.350	0.300	0.450	0.0	0.364	0.640	0.350	0.619	0.531	0.401	-
			Prec.	0.600	0.400	0.767	0.0	0.806	0.925	0.600	0.913	0.947	0.662	-
			F_1	0.442	0.343	0.567	0.0	0.502	0.757	0.442	0.738	0.680	0.499	-
			TP	*6.3*	*3.0*	*15.8*	*0.0*	*22.2*	*30.7*	*6.3*	*35.9*	*32.9*	-	*153.1*
word2vec	{1, 2}	256	*Rec.*	0.450	0.450	0.550	0.050	0.579	0.720	0.300	0.776	0.621	0.500	-
			Prec.	0.800	0.500	0.917	0.100	0.946	0.918	0.350	0.945	0.933	0.712	-
			F_1	0.576	0.474	0.688	0.067	0.718	0.807	0.323	0.852	0.746	0.587	-
			TP	*8.1*	*4.5*	*19.3*	*0.8*	*35.3*	*34.6*	*5.4*	*45.0*	*38.5*	-	*191.5*
fastText	{1, 2, 3}	256	*Rec.*	0.400	0.550	0.500	0.0	0.564	0.700	0.350	0.719	0.593	0.486	-
			Prec.	0.700	0.600	1.0	0.0	0.958	0.933	0.567	0.983	0.930	0.741	-
			F_1	0.509	0.574	0.667	0.0	0.710	0.800	0.433	0.831	0.724	0.587	-
			TP	*7.2*	*5.5*	*17.5*	*0.0*	*34.4*	*33.6*	*6.3*	*41.7*	*36.8*	-	*183.0*

We can compare the metrics in Table 4.12 with the Naïve Bayes equivalent found in Table 4.4 in Experiment 1. Similarly to the multi-label CNN, the binary CNN models seem to generally provide a modest improvement for the individual classification of each class label in comparison to the Naïve Bayes baselines. The average F_1-score for each NFR type binary CNN for each word embedding initialization provide a lift of 8.9-17.7% from the Naïve Bayes average.

4.3.3 Discussion

In Experiment 3, we run an exhaustive search for the optimal CNN models for multi-label NFR type classification. We then compare the performance of these CNN models with the Naïve Bayes metrics discussed in Section 4.1.3. In addition, we train individual binary CNNs for each NFR type and compare their performance with the Naïve Bayes equivalents.

Overall, the multi-label CNNs not only provide a general improvement to all baseline metrics, they also appear to work very well in improving the identification of under-represented class labels in an unbalanced multi-label classification problem. On the other hand, the binary CNNs for individual NFR types also seem to provide a slight general improvement across the board, but no outstanding trends were observed.

4.4 Experiment 4: Epoch Convergence

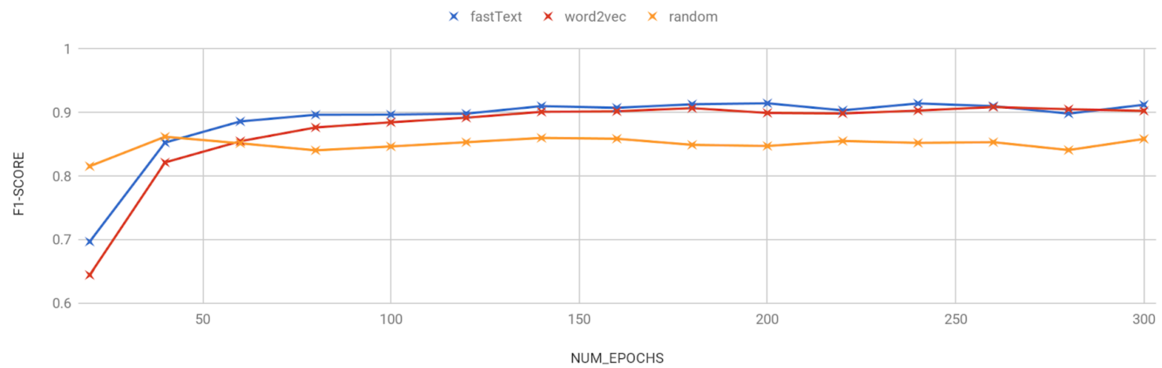
We need to validate that the choice to train on 140 epochs does not jeopardize performance. Recall that we arbitrarily chose 140 epochs through primitive diagnostics — choosing a number high enough to safely past the point of performance convergence but low enough to not excessively inflate the run time for each experiment.

We run this experiment on a subset of the primary classification problems to observe the performance trends applied to our two datasets. For SecReq-`SE` and NFR-`NF`, we run the optimal CNN models for each embedding initialization, from 20 epochs to 300 epochs with an interval of 20, to graph the effect of training epochs on F_1-score and cross-entropy loss.

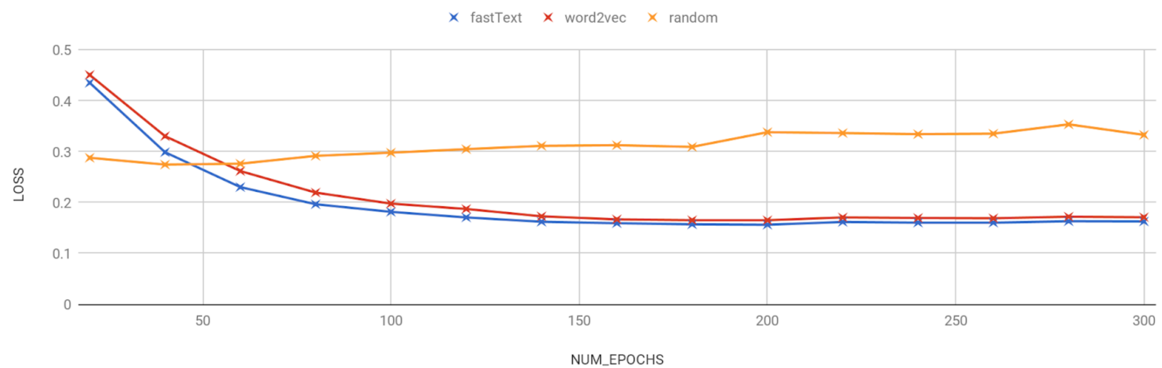
Figure 4.1 illustrates the F_1-score and cross-entropy loss for each embedding initialization over the domain of epochs for SecReq-`SE`. Likewise, Figure 4.2 serves the equivalent for NFR-`NF`.

4.4.1 SecReq-`SE`

The performance trends for SecReq-`SE` over 20 to 300 epochs, shown in Figure 4.1a and 4.1b, illustrate a very clear convergence of F_1-score and cross-entropy loss for the pre-trained word embedding (`word2vec` and `fastText`) trend lines. Figure 4.1a

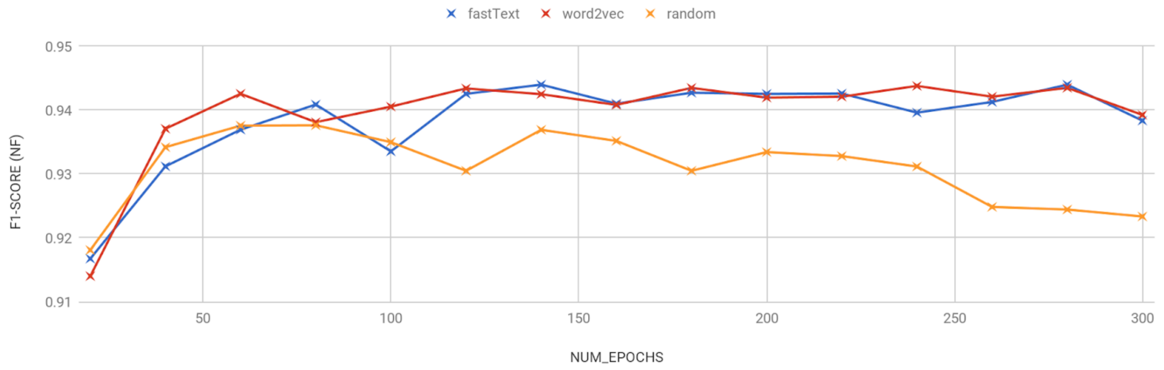


(a) # of Epochs vs. F_1-score

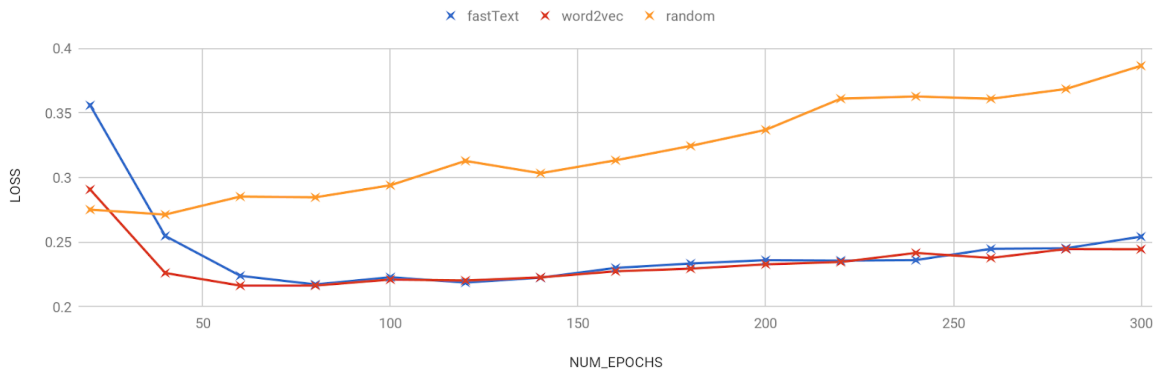


(b) # of Epochs vs. Loss

Figure 4.1: # of Epochs vs. Performance for SecReq-SE optimal CNN.



(a) # of Epochs vs. ΔLoss



(b) # of Epochs vs. Loss

Figure 4.2: # of Epochs vs. Performance for NFR-`NF` optimal CNN.

shows that F_1-scores for word2vec and fastText converge after 100 epochs, whereas random stays relatively level throughout. The word2vec and fastText trend lines both start off lower than random, but they both improve at nearly the same pace and supersede random after 60 epochs.

Similar observations can be observed for cross-entropy loss. Figure 4.1b shows that loss for word2vec and fastText, as expected, follow similar trend lines in contrast to random; the pre-trained embedding lines converge somewhere between 100 to 160 epochs.

An interesting observation is that for both metrics, the random trend lines in Figures 4.1a and 4.1b result in straight line rather than a curve, remaining relatively level throughout the entire domain. The pre-trained word embedding models, on the other hand, show a clear improvement in F_1-score and loss within 100 epochs.

4.4.2 NFR-NF

The performance for NFR-NF generally follow similar trends as SecReq-SE. Figure 4.2a illustrates the influence of training epochs on F_1-score, with word2vec and fastText once again following the same curve. The random trend line starts off aligned with the pre-trained word embedding models, but after 80 epochs, it starts to dip and fall far beneath the other two — a tell-tale sign of overfitting.

Figure 4.2b showcases interesting observations on the trend of cross-entropy loss. Similarly to SecReq-SE, the random trend line also appears linear. However, here it features a strong positive slope, supporting the observations of its declining F_1-score past 80 epochs. The pre-trained word embedding models once again follow the same trend; although fastText starts off slightly worse than word2vec, they produce nearly identical loss values past 80 epochs.

4.4.3 Discussion

Experiment 4 was designed to validate the selection of 140 epochs for training CNNs in Experiment 2 and beyond. In retrospect, not only should this experiment have been conducted for all classification problems defined in Table 4.1, it also should have been done before Experiment 2.

We want to train for a number of epochs high enough where performance is stabilized, but not too high as to avoid adding extraneous training time to each run. In addition, we want to avoid blatant overfitting, as observed with `random` in Figure 4.2b. Thus, we conclude that 140 epochs is a decent selection as the metrics from the two classification problems both converged at around 100 epochs.

4.5 Experiment 5: Cross-Dataset Security Requirements Classification

In an attempt to assess the quality of our classification models, we devise three cross-dataset security requirements classification problems utilizing SecReq-`SE` and NFR-`SE`:

1. *Train on SecReq-`SE` and validate on NFR-`SE`*, to gauge the quality of a security requirements classifier built from SecReq.
2. *Train on NFR-`SE` and validate on SecReq-`SE`*, to gauge the quality of a security requirements classifier built from NFR-`SE`.
3. *Hybrid dataset combining SecReq-`SE` and NFR-`SE`*, to gauge the performance of a security requirements classifier built from two different datasets.

10-fold cross-validation is applied to all three problems. For Problems 1 and 2 where the training and test sets come from different datasets, we perform cross-validation by training on 90% of the training set and validating on 10% of the test

set. To supplement those metrics, we train another classifier on the entire training set and validate on the entire test set. The following subsections detail the cross-dataset experiments conducted.

4.5.1 Train on SecReq-SE, Validate on NFR-SE

Our first assessment is to gauge how effective a security requirements classifier is when trained on SecReq and validated on a different dataset, in this case NFR-SE. Tables 4.13 and 4.14 tabulate the performance metrics for Naïve Bayes and our series of CNN classifiers (built with the optimal models for SecReq-SE, discussed in Section 4.2.1), run with and without 10-fold cross-validation.

Table 4.13: Results from NB and CNN classifiers trained on SecReq-SE, validated on NFR-SE using 10-fold cross-validation.

Classifier	*Embedding Type*	*Filter Sizes*	*# Filters per Size*	*Loss*	*Accuracy*	*Recall*	*Precision*	F_1*-score*
NB	-	-	-	-	0.759	**0.450**	0.365	0.403
CNN	random	$\{1, 2\}$	16	0.646	0.776	0.191	0.277	0.226
	word2vec	$\{1, 2\}$	128	0.482	**0.779**	0.410	**0.401**	**0.405**
	fastText	$\{1, 2, 3\}$	32	0.517	0.749	0.421	0.351	0.383

Table 4.14: Results from NB and CNN classifiers trained on SecReq-SE, validated on NFR-SE, without cross-validation.

Classifier	*Embedding Type*	*Filter Sizes*	*# Filters per Size*	*Loss*	*Accuracy*	*Recall*	*Precision*	F_1*-score*
NB	-	-	-	-	0.740	0.409	0.325	0.362
CNN	random	$\{1, 2\}$	16	0.575	0.787	0.258	0.370	0.304
	word2vec	$\{1, 2\}$	128	0.482	0.809	0.212	0.438	0.286
	fastText	$\{1, 2, 3\}$	32	0.359	**0.866**	**0.455**	**0.700**	**0.551**

As observed in Table 4.13, all classifiers run with 10-fold cross-validation perform with F_1-scores $\leq 40\%$, half the score of our Naïve Bayes and CNN classifiers trained and validated on NFR-SE alone. However, the Naïve Bayes baseline for this experi-

ment performs at comparable levels with CNNs (with the exception of `random`, which performs much worse). This suggests that the nature of the datasets, whether it be the quality of SecReq in training or the nature of NFR-SE when testing with such a large k, is likely the culprit rather than the different classifiers we employ.

Table 4.14 display the results from this classification problem, except without cross-validation. We train on the entire SecReq dataset and test on the entire NFR-SE dataset. The metrics suggest similar results, with the exception of the `fastText`-powered CNN achieving double the precision of the Naïve Bayes baseline.

4.5.2 Train on NFR-SE, Validate on SecReq-SE

Our second assessment is to gauge how effective a security requirements classifier is when trained on NFR-SE and validated on a different dataset, in this case SecReq. Tables 4.15 and 4.16 tabulate the performance metrics for Naïve Bayes and our series of CNN classifiers (built with the optimal models for NFR-SE, discussed in Section 4.2.3), run with and without 10-fold cross-validation.

Table 4.15: Results from NB and CNN classifiers trained on NFR-SE, validated on SecReq, using 10-fold cross-validation.

Classifier	*Embedding Type*	*Filter Sizes*	*# Filters per Size*	*Loss*	*Accuracy*	*Recall*	*Precision*	*F_1-score*
NB	-	-	-	-	**0.673**	0.246	0.612	0.351
CNN	random	$\{1\}$	128	1.494	0.663	0.139	**0.618**	0.227
	word2vec	$\{1, 2\}$	32	0.762	0.656	0.335	0.522	0.408
	fastText	$\{1\}$	64	0.723	0.655	**0.430**	0.524	**0.472**

As expected, our results for this experiment in Table 4.15 follow similar trends to its converse discussed above in Section 4.5.1. For 10-fold cross-validation, the highest F_1-scores still fall less than half of its performance from our Naïve Bayes and CNN classifiers trained and validated on SecReq-SE alone. It seems that the incorporation of pre-trained word embeddings help boost recall 8.9-18.4% above the Naïve Bayes

Table 4.16: Results from NB and CNN classifiers trained on NFR-SE, validated on SecReq, without cross-validation.

Classifier	*Embedding Type*	*Filter Sizes*	*# Filters per Size*	*Loss*	*Accuracy*	*Recall*	*Precision*	*F_1-score*
NB	-	-	-	-	0.685	0.263	0.644	0.373
CNN	random	$\{1\}$	128	1.491	0.671	0.145	**0.684**	0.240
	word2vec	$\{1, 2\}$	32	0.722	0.673	0.358	0.566	0.438
	fastText	$\{1\}$	64	0.576	**0.737**	**0.676**	0.621	**0.647**

baseline and 19.6-29.1% above the random CNN configuration.

Table 4.16 display the results from this classification problem, except without cross-validation. We train on the entire NFR-SE dataset and test on the entire SecReq dataset. The experiment results in fairly close results to cross-validation, with the same exception of the fastText-powered CNN achieving more than double the recall of the Naïve Bayes baseline.

4.5.3 Hybrid Security Dataset

Our final cross-dataset assessment involves creating a hybrid security requirements dataset by combining both SecReq-SE and NFR-SE. The hybrid dataset contains 803 total requirements, 245 of which are security-related; 76% of the security-related requirements from SecReq and 24% from NFR.

Table 4.17 tabulate the performance metrics for Naïve Bayes and our series of CNN classifiers (built with both the optimal models for SecReq-SE and NFR-SE, discussed in Sections 4.2.1 and 4.2.3), run with 10-fold cross-validation.

In comparison to the Naïve Bayes performance, the CNNs provide a 5.8-10.1% lift in precision. We observe that the random embedding CNNs already offer the improvements in precision, suggesting that the employment of CNNs on this hybrid dataset is responsible for improvements in precision with the expense of recall. However,

Table 4.17: Results from NB and CNN classifiers trained and validated on security hybrid dataset, using 10-fold cross validation.

Classifier	*Embedding Type*	*Filter Sizes*	*# Filters per Size*	*Loss*	*Accuracy*	*Recall*	*Precision*	F_1*-score*
NB	-	-	-	-	0.893	**0.824**	0.807	0.815
CNN	random	$\{1,2\}$	16	0.398	0.887	0.689	0.886	0.776
		$\{1\}$	128	0.394	0.896	0.727	0.891	0.801
	word2vec	$\{1,2\}$	128	0.262	0.915	0.787	0.901	**0.840**
		$\{1,2\}$	32	0.243	0.902	0.784	0.865	0.822
	fastText	$\{1,2,3\}$	32	0.219	**0.916**	0.780	**0.908**	0.839
		$\{1\}$	64	0.237	0.911	0.799	0.880	0.838

it appears that the addition of pre-trained word embeddings help the CNN models counteract the fall in recall.

4.5.4 Discussion

Experiment 5 is a surface level attempt at assessing the prospect of our CNN models overfitting to a specific dataset. As our datasets are comprised of very domain-specific vocabulary and writing styles, we suspect that there is a strong potential for overfitting.

Our results from Section 4.5.1 and 4.5.2 show our CNN models performing either just as poorly as Naïve Bayes or slightly better in either recall or precision. This suggests that the general nature of training a classifier on word features on one of these datasets and applying the model on a completely different one is difficult as the classifier learns dataset-specific trends. Hence, whatever performance boost from our CNN models in comparison to Naïve Bayes can be assumed to be independent of overfitting.

Chapter 5

RELATED WORK

During the timeframe of this thesis, we submitted our findings from our initial trial of word2vec and TensorFlow applications on requirements classification to the proceedings of the *RE Data Challenge* posed in the *25th IEEE International Requirements Engineering Conference* (RE '17) [17]. Alongside us, a number of other researchers also tackled the same focus area of requirements classification, exploring new preprocessing, sampling, and modeling strategies. The following list previews these sister papers from RE '17:

- Abad et al. [9] questioned how "grammatical, temporal, and sentimental characteristics of a sentence" impact the classifying functional vs. non-functional requirements in the NFR dataset. They also evaluated the performance of various machine learning algorithms on NFR type classification.

- Kurtanović et al. [29] questioned whether various sampling strategies can successfully handle class imbalance and improve performance of requirements classification.

- Munaiah et al. [37] investigated whether a "domain-independent classifier can effectively identify security requirements across domains".

5.1 Preprocessing Strategies

Abad et al. evaluated the effects of the following preprocessing rules to standardize the requirements text [9]:

- **Part-of-speech (POS) Tagging.** Tag each word in each requirement with their POS (i.e., noun, verb, adjective).

- **Entity Tagging.** Define a dictionary of context-based products and users that can be assigned a general entity name (e.g., `"realtor"` ⇒ `"USER"`).

- **Temporal Tagging.** Normalize all time-telling expressions into general types of temporal objects (e.g., `"6pm"` ⇒ `"TIME"`, `"2 minutes"` ⇒ `"DURATION"`).

- **Co-occurrence and Regular Expression Replacements.** Replace regular expressions of common POS patterns and words related to keywords with the keyword itself.

The requirements were first cleaned of formatting errors and encoding. The part-of-speech were then assigned, prompting for the extraction of syntactic and keyword features. A final feature set was assembled for both the unprocessed and processed data, which were then used to train a C4.5 decision tree for `NF` vs. `F` requirements classification for both datasets. Both classifiers were evaluated with 10-fold cross-validation.

The results from the experiments show that the additional preprocessing steps significantly improve the accuracy of the decision tree classifier from 89.92% to 94.40% and F_1-score from 90% to 94%. Refer to Table 5.1 below for exact measurements.

5.1.1 Comparison

Table 5.1 displays the results from Abad et al., comparing the performance boost from their preprocessing methods against the unprocessed dataset. The table also showcases our CNN and word embedding approaches on NFR-`NF` (retrieved from Table 4.7), the same classification problem investigated by Abad et al. We observe that

our researched methods score comparably in F_1-score and also beat the unprocessed baseline in recall.

Table 5.1: NFR-`NF` binary classification results from Abad et al. [9] compared to our CNN word embedding approaches.

Method		*Recall*	*Precision*	*F_1-score*
Abad et al. [9]	Unprocessed	0.88	0.95	0.91
	Processed	0.93	**0.98**	**0.95**
`CNN`	`random`	**0.96**	0.92	0.94
	`word2vec`	0.94	0.93	0.94
	`fastText`	**0.96**	0.93	**0.95**

5.2 Sampling Strategies

Using the NFR dataset, Kurtanović et al. ran a series of experiments with SVM-based classifiers to test the effects of the following sampling strategies [29]:

- **Undersampling.** The majority class is *undersampled*, or reduced, to achieve a balanced distribution.
- **Oversampling.** The minority class is *oversampled*, or supplemented, with additional samples acquired from another dataset to achieve a balanced distribution.

The additional dataset they had on hand was the UC dataset, a requirements dataset with usability and performance requirements derived from a sample of crawled Amazon software user comments (UC). The team thus investigated the effects of under- and oversampling on classifying usability (`US`) and performance (`PE`) requirements.

The requirements were first normalized, stripped of stop words, and lemmatized. Two types of experiments were conducted, one that emphasized minority class rarity and another that expanded the minority class. For the former, the minority class was undersampled for the training set and then oversampled by filling it with a random subsample from the UC dataset. For the latter, the entire minority class sample from the NFR dataset was used and oversampled with additional examples from the UC dataset. For both types of experiments, the final training set had an equal ratio of both classes. A SVM binary classifier for either `US` and `PE` were trained for all experiments and ran with 10-fold cross-validation.

For the first experiment, the results show that using just the NFR dataset and using the UC dataset yields similar classification performance. For the second experiment, the results show that oversampling the minority class with the UC dataset does not significantly improve classification performance. Refer to Table 5.2 below for exact measurements.

5.2.1 Comparison

Table 5.2 showcases the results from the research from Kurtanović et al. on under- and over-sampling using the NFR and UC datasets on binary classification of `US` and `PE` requirements. The table also compares their results with our CNN word embedding models on NFR-`US` and NFR-`PE` (retrieved from Table 4.12). Judging based on the metrics, it appears as though their oversampling method improves precision for NFR-`PE`, but at the expense of recall, thus resulting in no improvement in overall F_1-score. Furthermore, our deep learning approaches perform comparably to the oversampling metrics, only beating their baseline in precision.

Table 5.2: NFR-`US` and NFR-`PE` binary classification results from Kurtanović et al. [29] compared to our CNN word embedding approaches.

Method				`US`			`PE`		
		NFR Sample	**UC Sample**	*Rec.*	*Prec.*	F_1	*Rec.*	*Prec.*	F_1
Kurtanović et al. [29]	Baseline	33% $C1_{min}$	-	0.80	0.60	0.69	0.66	0.70	0.68
		66% $C1_{min}$	-	**0.85**	0.80	0.82	**0.88**	0.89	**0.88**
	Oversampling	33% $C1_{min}$	66% $C1_{min}$	0.66	0.58	0.62	0.52	**0.93**	0.67
		66% $C1_{min}$	33% $C1_{min}$	0.83	0.83	**0.83**	0.74	**0.93**	0.82
`CNN`	`random`			0.54	**0.95**	0.68	0.64	**0.93**	0.76
	`word2vec`			0.62	0.93	0.75	0.72	0.92	0.81
	`fastText`			0.54	0.93	0.72	0.70	**0.93**	0.80

5.3 Domain-Independent Model and One-Class SVMs

Although classifiers work best when trained and applied on domain-specific data [26], domain-specific datasets are not readily available. Because of this, Munaiah et al. acquired a domain-independent dataset called the Common Weakness Enumeration (CWE) to evaluate the efficacy of domain-independent classifiers in identifying security requirements across domains [37]. Although CWE is not a requirements dataset, it does contain security-related text.

Munaiah et al. wanted to assess the performance of one-class classification for security requirements identification trained on a domain-independent dataset. In order to do this, they built a *One-Class SVM*, trained it on TF-IDF vectorized documents from the CWE dataset, and tested it on the individual SRS documents in the SecReq dataset (i.e., `CPN`, `ePurse`, `GPS`). Their results show that the One-Class SVM beat the Naïve Bayes baseline from Knauss et al. [26], yielding a 16% boost in the average F_1-score. Refer to Table 5.3 for exact measurements.

5.3.1 Comparison

Table 5.3 tabulates the performance comparison between the Naïve Bayes baselines by Knauss et al., One-Class SVM approach by Munaiah et al., and our best CNN models on SecReq single domain classification. In their paper, Munaiah et al. misreported the baseline metrics from Knauss et al., confusing recall and precision. Thus, we can only evaluate the F_1-scores from their experiments.

The table showcases the best results from validating on one SRS document while training on a different document. For `CPN` and `GPS`, our approach performs comparably with the One-Class SVM approach by Munaiah et al. However, for `ePurse`, our approach scores a significant 21% higher in F_1-score.

Table 5.3: Comparison of most effective security requirements classifiers on single domain evaluations between Knauss et al. [26], Munaiah et al. [37], and our CNN word embedding approaches.

Validation	*Method*	*Training*	*F_1-score*
`CPN`	Naïve Bayes *(Knauss et al. [26])*	`GPS`	*0.40*
	One-Class SVM *(Munaiah et al. [37])*	CWE	**0.74**
	`CNN + word2vec`	`GPS`	0.73
`ePurse`	Naïve Bayes *(Knauss et al. [26])*	`GPS`	*0.58*
	One-Class SVM *(Munaiah et al. [37])*	ExCWE	0.61
	`CNN + fastText`	`GPS`	**0.82**
`GPS`	Naïve Bayes *(Knauss et al. [26])*	`ePurse`	*0.57*
	One-Class SVM *(Munaiah et al. [37])*	ExCWE	**0.68**
	`CNN + word2vec/fastText`	`ePurse`	0.67

5.4 Topic Modeling, Clustering, and Binarized Naïve Bayes

Abad et al. compared the performance of various machine learning algorithms on NFR type classification [9]. The methods explored were topic modeling, clustering, and Naïve Bayes.

Topic modeling and *clustering* are both unsupervised learning techniques that attempts to categorize unlabelled documents or text into groups. The topic modeling algorithms they employed were *Latent Dirichlet Allocation* (LDA) and *Biterm Topic Modeling* (BTM), and the clustering algorithms explored were *Hierarchical Agglomerative* and *K-means*. They also used a variation of multinomial Naïve Bayes called *Binarized Naïve Bayes* (BNB) that utilizes word presence rather than frequency.

Each technique was used to train a multi-label classifier for identifying NFR types in the NFR dataset. As an extension to their preprocessing experiments discussed in Section 5.1, Abad et al. trained each classifier on the processed dataset as well as the unprocessed dataset. Each experiment was validated with 5-fold cross-validation.

To summarize their findings, they concluded that (1) their preprocessing approach significantly improved the performance of all classification methods (e.g., LDA and BNB doubled in precision and recall), (2) BNB scored the highest performance, and (3) BTM did not perform well in this task.

Chapter 6

CONCLUSIONS AND FUTURE WORK

Document classification is a common machine learning problem, finding itself at the intersection between disciplines of natural language processing and artificial intelligence. With the growing popularity and availability of deep learning, it is natural to consider deep learning for document analysis tasks. However, these methodologies are usually reserved for large-scale problems with large volumes of feature-rich data samples.

With that said, we are curious about the feasibility of a subset of deep learning methodologies applied to an unconventional environment: software requirements document analysis. Software requirements documents are in nature short in length, often comprised of a couple of sentences. In addition, the requirements datasets only contain hundreds to thousands of documents, which is orders of magnitude less in volume than typically deemed necessary for deep learning.

In this thesis, we investigate these concerns by exploring the efficacy of convolutional neural networks in training classifiers for various classification problems extracted from two well-studied software requirements datasets. In conjunction, we vectorize our requirements documents with word embeddings to explore whether pre-trained word embeddings can supplement our documents with semantic features. We measure the impact of these two concepts by comparing the performance with baseline metrics acquired from training Naïve Bayes classifiers on word count features, a standard machine learning approach also employed by Knauss et al. when studying security requirements classification [26].

Table 6.1: Summary of NB baseline and CNN performance for binary classification problems, trained with 140 epochs and run with 10-fold cross-validation.

Problem	*Classifier*	*Embedding Type*	*Recall*	*Precision*	*F_1-score*
SecReq-SE	NB	-	*0.888*	*0.791*	*0.837*
	CNN	random	0.815	0.880	0.846
		word2vec	0.877	**0.919**	0.897
		fastText	**0.915**	0.907	**0.911**
	Max Improvement		*+0.027*	*+0.128*	*+0.074*
NFR-NF	NB	-	*0.926*	*0.915*	*0.921*
	CNN	random	0.959	0.918	0.938
		word2vec	0.943	**0.929**	0.936
		fastText	**0.962**	0.929	**0.945**
	Max Improvement		*+0.036*	*+0.014*	*+0.024*
NFR-SE	NB	-	*0.638*	*0.958*	*0.766*
	CNN	random	0.593	**0.957**	0.732
		word2vec	0.743	0.929	0.826
		fastText	**0.774**	0.953	**0.854**
	Max Improvement		*+0.136*	*–0.001*	*+0.088*

Table 6.2: Summary of NB baseline and CNN F_1-scores for NFR-Types multi-label classification, trained with 140 epochs and run with 10-fold cross-validation.

Classifier	*Embedding Type*	A	L	LF	MN	O	PE	SC	SE	US	*Avg.*
NB	-	*0.297*	*0.0*	*0.654*	*0.0*	*0.690*	*0.779*	*0.375*	*0.843*	*0.708*	*0.497*
CNN	random	0.692	0.574	0.750	0.200	0.728	0.787	0.565	0.845	0.764	0.657
	word2vec	**0.804**	**0.747**	0.771	**0.533**	**0.808**	**0.822**	**0.680**	**0.873**	**0.865**	**0.772**
	fastText	0.774	0.732	**0.835**	0.444	0.799	0.809	0.616	0.860	0.816	0.745
Max Improvement		*+0.507*	*+0.747*	*+0.181*	*+0.533*	*+0.118*	*+0.054*	*+0.305*	*+0.030*	*+0.157*	*+0.275*

Table 6.3: Summary of NB baseline and CNN TP count for NFR-Types multi-label classification, trained with 140 epochs and run with 10-fold cross-validation.

Classifier	*Embedding Type*	A	L	LF	MN	O	PE	SC	SE	US	*Total*
NB	-	*4.5*	*0.0*	*18.4*	*0.0*	*49.1*	*37.4*	*5.4*	*50.0*	*55.4*	*220.2*
CNN	random	12.3	6.0	27.2	3.2	43.1	36.6	10.8	48.9	45.4	233.4
	word2vec	**15.6**	**8.0**	28.5	**8.3**	**45.4**	**40.7**	**15.3**	49.6	52.9	**264.3**
	fastText	**15.6**	7.0	**29.8**	7.2	45.2	39.7	14.4	**49.7**	**53.3**	262.0
Max Improvement		*+11.1*	*+8.0*	*+11.4*	*+8.3*	*–3.7*	*+3.3*	*+9.9*	*–0.03*	*–2.1*	*+44.1*

We staged a battery of experiments to evaluate our research questions, using the classifier training system we built using TensorFlow, Scikit-Learn, Gensim and other tools. Although the magnitude of improvement ranges from problem to problem, we can observe a positive impact attributed to our CNNs, especially from those powered with pre-trained word embeddings. For example, as shown in Table 6.1, our SecReq security requirements CNN equipped with `fastText` accomplish an generous 7.4% boost in F_1-score, attributed the impressive 12.8% boost in precision. On the other hand, because our baseline classifier for NFR-`NF` already scores very high, our NFR non-functional requirements CNNs can only contribute a modest lift of just a few percentage points.

We observe that the performance of the CNNs with randomly initialized word embeddings (`random`) hover around baseline measures, and the incorporation of `word2vec` or `fastText` are responsible for the more significant boost in performance. Furthermore, it appears that the `fastText` embeddings perform slightly better in the binary classification problems, whereas `word2vec` performs mildly better for the multi-label `NF` type classification problem. These nuances can either be attributed to the intricacies of the models in which these embeddings were trained with and/or the corpora they were trained on.

Overall, we conclude that utilizing word embeddings in document vectorization when training CNNs provide a modest improvement in the domain of software requirements classification. However, we speculate that the measurement of performance boost is potentially diminished because of the high performance benchmarks from the baseline methods. Thus, our studied methods might be more useful applied to datasets and problems where other approaches are not as successful.

6.1 Future Work

The work of this thesis merely scrapes the surface of the research opportunities in this field. Aside from exploring more datasets, there are numerous avenues left unexplored in our methods and validation.

6.1.1 One-vs.-All Classification

As mentioned in Section 4.1.3, we considered comparing a single multi-label classifier with the *one-vs.-all* strategy for the classification of the nine types of non-functional requirements, as the one-vs.-all strategy was employed by prior research for the NFR dataset [11]. Unfortunately due to the timeline of this thesis, we were unable to implement support for this evaluation in our system.

6.1.2 CNN Optimization

We focused on filter sizes, filter count, and number of training epochs when configuring our CNNs, leaving out a whole expanse of CNN hyperparameters. For example, the dropout and l_2 norm factors are two parameters that were left untapped in our experiments. In addition, there are more sophisticated methods to optimizing CNNs for small datasets besides exhaustive trial-and-error that we did not consider [45].

We should also investigate more on the implications of the varying neural network shapes, namely the benefits and tradeoffs of the filter sizes and filter volume. We observe from our results from Experiments 2 and 3 that the optimal CNN models often include filters of size 1, either alone or in conjunction with other sizes. Filters of size 1 harbor unigrams within the documents, which in turn reduces the effect of locality we expect to utilize. This triggers the question of what we are gaining from using such a heavy structure versus more lightweight approaches.

6.1.3 Word Embeddings

Aside from neural network specific tuning, we also in the future try to stretch the usage of word embeddings and fill in the holes that we left agape. For example, we can supplement fastText with word2vec embeddings and vice versa if a vocabulary word is absent in one model but present in another. We can further uncover the raw influence of pre-trained word embeddings by vectorizing the documents with them and feeding them into traditional machine learning classifiers such as Naïve Bayes and SVMs.

To better compare the influence of the embedding models, we should consider training a word2vec model on the Wikipedia corpus that our current fastText model is based on and vice versa with the Google News corpus. Currently with the two embedding models being trained on two different corpora, it is difficult to confidently declare that the a word embedding model is better suited for a certain task because of how it was trained versus what corpus it was trained on. Another avenue would be to train word2vec and fastText models on a software specific corpus. However, this might be impossible without industry grade power, especially if the corpora are excessively large (as with Wikipedia and Google News).

6.1.4 FastText

Lastly, with the fast-paced technology industry always publishing new and improved tools to the public, we have witnessed numerous advancements and additions to the methods we employed during the span of our research. Mid-way through our efforts, we discovered Facebook's fastText but only had the capacity to employ their Wikipedia-trained word embeddings. Since then, Facebook has released a new set of fastText embeddings pre-trained on the Common Crawl corpus. In addition, as briefed in Section 2.4.2, the fastText bundle includes a text classifier that trains much faster than deep neural networks without relinquishing performance quality. Future research should explore the fastText text classifier and compare its performance as well as training time metrics with our CNN outputs.

BIBLIOGRAPHY

[1] FastText. https://fasttext.cc/.

[2] Gensim. https://pypi.python.org/pypi/gensim.

[3] Natural Language Processing Toolkit. http://www.nltk.org/.

[4] Quality Attributes (NFR) dataset. http://openscience.us/repo/requirements/requirements-other/nfr.html.

[5] Scikit-Learn. http://scikit-learn.org/.

[6] SecReq dataset. http://www.se.uni-hannover.de/pages/en:projekte_re_secreq.

[7] TensorFlow. https://www.tensorflow.org/.

[8] Word2vec. https://code.google.com/archive/p/word2vec/.

[9] Z. S. H. Abad, O. Karras, P. Ghazi, M. Glinz, G. Ruhe, and K. Schneider. What works better? a study of classifying requirements. In *2017 IEEE 25th International Requirements Engineering Conference (RE)*, RE '17, pages 496–501. IEEE Computer Society, 2017.

[10] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *CoRR*, abs/1607.04606, 2016.

[11] A. Casamayor, D. Godoy, and M. Campo. Identification of non-functional requirements in textual specifications: A semi-supervised learning approach. *Inf. Softw. Technol.*, 52(4):436–445, Apr. 2010.

[12] L. Chung and B. A. Nixon. Dealing with non-functional requirements: Three experimental studies of a process-oriented approach. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 25–37, New York, NY, USA, 1995. ACM.

[13] M. Claesen and B. D. Moor. Hyperparameter search in machine learning. *CoRR*, abs/1502.02127, 2015.

[14] J. Cleland-Huang, R. Settimi, X. Zou, and P. Solc. Automated classification of non-functional requirements. *Requir. Eng.*, 12(2):103–120, May 2007.

[15] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12:2493–2537, Nov. 2011.

[16] A. Dekhtyar, O. Dekhtyar, J. Holden, J. H. Hayes, D. Cuddeback, and W. Kong. On human analyst performance in assisted requirements tracing: Statistical analysis. In *RE 2011, 19th IEEE International Requirements Engineering Conference, Trento, Italy, August 29 2011 - September 2, 2011*, pages 111–120, 2011.

[17] A. Dekhtyar and V. Fong. RE Data Challenge: Requirements identification with Word2Vec and TensorFlow. In *2017 IEEE 25th International Requirements Engineering Conference (RE)*, RE '17, pages 484–489. IEEE Computer Society, 2017.

[18] J. Eckhardt, A. Vogelsang, and D. M. Fernández. Are "non-functional" requirements really non-functional?: An investigation of non-functional requirements in practice. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 832–842, New York, NY, USA, 2016. ACM.

[19] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[20] J. H. Hayes and A. Dekhtyar. Humans in the traceability loop: Can't live with 'em, can't live without 'em. In *Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE '05, pages 20–23, New York, NY, USA, 2005. ACM.

[21] N. Indurkhya and F. J. Damerau. *Handbook of Natural Language Processing*. Chapman & Hall/CRC, 2nd edition, 2010.

[22] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov. Bag of tricks for efficient text classification. *CoRR*, abs/1607.01759, 2016.

[23] Y. Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1746–1751, 2014.

[24] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[25] E. Knauss, S. H. Houmb, S. Islam, J. Jürjens, and K. Schneider. Eliciting security requirements and tracing them to design: An integration of common criteria, heuristics, and UMLsec. *Requirements Engineering*, 15(1):63–93, Mar. 2010.

[26] E. Knauss, K. Schneider, S. Houmb, I. Shareeful, and J. Jürjens. Supporting requirements engineers in recognising security issues. In *Proceedings of 17th Intl. Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ'11)*. Springer, 2011.

[27] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[29] Z. Kurtanović and W. Maalej. Automatically classifying functional and non-functional requirements using supervised machine learning. In *2017 IEEE 25th International Requirements Engineering Conference (RE)*, RE '17, pages 490–495. IEEE Computer Society, 2017.

[30] Q. V. Le and T. Mikolov. Distributed representations of sentences and documents. In *ICML*, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 1188–1196. JMLR.org, 2014.

[31] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, volume 86, pages 2278–2324, 1998.

[32] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2nd edition, 2014.

[33] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[34] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[35] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems*, NIPS'13, pages 3111–3119, USA, 2013. Curran Associates Inc.

[36] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[37] N. Munaiah, A. Meneely, and P. K. Murukannaiah. A domain-independent model for identifying security requirements. In *2017 IEEE 25th International Requirements Engineering Conference (RE)*, RE '17, pages 506–511, 2017.

[38] K. O'Shea and R. Nash. An introduction to convolutional neural networks. 11 2015.

[39] A. Rashwan, O. Ormandjieva, and R. Witte. Ontology-based classification of non-functional requirements in software specifications: A new corpus and SVM-based classifier. In *Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference*, COMPSAC '13, pages 381–386, Washington, DC, USA, 2013. IEEE Computer Society.

[40] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[41] F. Sebastiani. Machine learning in automated text categorization. *ACM Comput. Surv.*, 34(1):1–47, Mar. 2002.

[42] F. Shaheen, B. Verma, and M. Asafuddoula. Impact of automatic feature extraction in deep learning architecture. In *2016 International Conference on Digital Image Computing: Techniques and Applications (DICTA)*, pages 1–8, Nov 2016.

[43] I. Sommerville and P. Sawyer. *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1997.

[44] M. J. Zaki and W. M. Jr. *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press, New York, NY, USA, 2014.

[45] Y. Zhang and B. C. Wallace. A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. *CoRR*, abs/1510.03820, 2015.

APPENDICES

Appendix A

TF-IDF EXPERIMENTS

Table A.1: Naïve Bayes (with `TfidfVectorizer`) results for SecReq-`SE` binary classification.

Method	*Recall*	*Precision*	*F_1-score*
`MultinomialNB`	0.637	0.878	0.738
Knauss et al. [26]	*0.91*	*0.78*	*0.84*

Table A.2: Naïve Bayes (with `TfidfVectorizer`) results for NFR-`NF` binary classification.

Requirement Type	*Recall*	*Precision*	*F_1-score*
`NF`	0.946	0.878	0.911
`F`	0.807	0.917	0.858
Average	*0.877*	*0.898*	*0.885*

Table A.3: Naïve Bayes (with `TfidfVectorizer`) results for NFR-Types multi-label classification.

Metric	`A`	`L`	`LF`	`MN`	`O`	`PE`	`SC`	`SE`	`US`	*Average*	*Total*
Recall	0.150	0.0	0.233	0.0	0.724	0.763	0.0	0.907	0.921	0.411	-
Precision	0.300	0.0	0.450	0.0	0.573	0.901	0.0	0.653	0.504	0.376	-
F_1-score	0.200	0.0	0.307	0.0	0.639	0.826	0.0	0.759	0.652	0.393	-
TP	*2.7*	*0.0*	*8.2*	*0.0*	*44.2*	*36.6*	*0.0*	*52.6*	*57.1*	-	*201.4*