Automatic Multi-Class Non-Functional Software Requirements Classification Using Neural Networks

Cody Baker, Lin Deng, Suranjan Chakraborty and Josh Dehlinger
Department of Computer and Information Sciences
Towson University, Towson, MD, USA
cbaker18@students.towson.edu; {ldeng, schakraborty, jdehlinger}@towson.edu

Abstract—Advances in machine learning (ML) algorithms, graphics processing units, and readily available ML libraries have enabled the application of ML to open software engineering challenges. Yet, the use of ML to enable decision-making during the software engineering lifecycle is not well understood as there are various ML models requiring parameter tuning. In this paper, we leverage ML techniques to develop an effective approach to classify software requirements. Specifically, we investigate the design and application of two types of neural network models, an artificial neural network (ANN) and a convolutional neural network (CNN), to classify non-functional requirements (NFRs) into the following five categories: maintainability, operability, performance, security and usability. We illustrate and experimentally evaluate this work through two widely used datasets consisting of nearly 1,000 NFRs. Our results indicate that our CNN model can effectively classify NFRs by achieving precision ranging between 82% and 94%, recall ranging between 76% and 97% with an F-score ranging between 82% and 92%.

Keywords—non-functional requirements, requirements engineering, machine learning

I. Introduction

When developing software, the first step is to collect, understand, and specify software requirements (i.e., requirements engineering) [1]. Software requirements significantly impact the success of a software project because all subsequent development activities (e.g., design, implementation, and testing) depend on precise requirements. Requirements are used to verify and validate the completion and success of software systems. Prior research indicates that up to 80% of the issues leading to customer dissatisfaction with delivered software are due to problems originating during requirements engineering [2]. Software requirements can be categorized into two types: functional requirements (FRs) and non-functional requirements (NFRs). As NFRs are constraints within different categories, they must be processed separately and classified accurately. For example, performance NFRs derive performance tests of the system, security NFRs lead to security mechanisms, measures, and policies, maintenance NFRs help outline maintenance plan. Inappropriate handling of NFRs can increase software maintenance costs and impact software quality [3]. Therefore, the identification, and classification of NFRs becomes a critical and necessary step.

As most software requirements are written in natural language, the process of analyzing requirements, especially NFRs, has not been fully automated. Software architects, analysts, and developers always find it challenging and time-consuming to analyze, understand, and classify NFRs, because all of these tasks require expertise, training, experience, and domain knowledge [4]. NFRs written in natural language can be vague, ambiguous, and/or imprecise leading to software engineer's misunderstanding of what is required of the software to be developed [5][4]. People with different education backgrounds, culture, or expertise may understand the same NFR differently [1]. This is understandably an issue as making sense of the nature of NFRs is important for finalizing specifications and the subsequent design model. Also, there always exists conflicts, interactions, and/or dependencies amongst different NFRs, which further complicates the whole requirements engineering process [6]. Human errors and mistakes are highly likely to be introduced and injected into software artifacts when performing complicated manual tasks such as NFR classification. The previous work of Ameller et al. [5] found that there was no automated tool to support the analysis and management of NFRs. These problems can lead to severe long-term consequences, such as budget overrun, quality and security issues, and customer dissatisfaction [3].

Recently, machine learning (ML) algorithms, tools, and techniques have been leveraged to tackle challenges that have never been solved before [7], [8]. This paper aims to design an effective approach that can automatically classify NFRs into multiple categories. Specifically, we design two kinds of neural network models (i.e., artificial neural network (ANN) and convolutional neural network (CNN)) to accommodate the classification of NFRs. We conduct an experimental investigation of applying these two ML techniques to "understand" and classify NFRs automatically. In this study, due to the limited availability of pre-labeled data categories (classes) in our dataset, we select the following five categories of NFRs: maintainability, operability, performance, security, and usability. However, our models can be universally applied to understand, process, and classify NFRs in any categories.

Our experimental results indicate that our CNN model can effectively classify NFRs into five different categories, while achieving precision ranging between 82% and 94%, recall ranging between 76% and 97% with an F-score ranging between 82% and 92%. An existing NFR dataset from the International Requirements Engineering Conference's 2017 Data Challenge dataset [9] indicates that five security experts failed to unanimously classify more than 50% of security requirements. This indicates that our approach significantly outperforms the manual work of specialists.

In summary, the contributions of this paper are:

- The design and implementation of two neural network models that are able to classify NFRs automatically.
- The experimental evaluation of these two models with two widely used datasets of software requirements and reporting of performance results.

The proposed ML approach is the first and foremost step of the long-term goals of the research in applying ML in requirements engineering. These goals include: 1. to significantly reduce human efforts and save software development costs by automating the analysis process of NFRs; and 2. to reduce human manual errors, mistakes and misunderstandings in requirements engineering process, which can further reduce potential requirement-related defects and bugs in software systems. Further, classification offers a mechanism to reduce cognitive effort in analyzing NFRs for developing down-stream artifacts. Consequently, development of a method that can automate this efficiently has significant practical implications. The work presented here is part of a larger effort to incorporate ML techniques into solving software engineering problems.

The rest of this paper is organized as follows. Section II presents the general approach designed in our research. Section III presents the experiments used to evaluate our approach and an analysis of the experimental results. Section IV gives an overview of related research, and the paper concludes and suggests future work in Section V.

II. General Approach

This section introduces the general approach of our neural network models for classifying NFRs.

A. *Categories of NFRs*

For over three decades, software engineering research has designed different methods for categorizing NFRs. For example, Roman [10] proposed six classes of NFRs: interface, performance, operating, life-cycle, economic, and political. Sommerville introduced three major groups of NFRs: product requirements, organizational requirements, external requirements [1]. Chung et al. [6] provided the most comprehensive categorization for NFRs, which includes more than 150 different classes. A different set of 14 categories was described by Slankas and Williams [11]. Therefore, it is likely that, depending on the expertise and domain knowledge of the analyst, an NFR may be classified into more than one type. Unavoidably, this increases the uncertainty in NFR analysis during requirements engineering and brings additional challenges to software architects, analysts, and developers. The datasets used in this research have 14 categories, which will be introduced in Section III.A.

B. *Artificial Neural Network (ANN) Model*

Figure 1 shows the structure of the ANN model used in this research. The leftmost layer is the input layer. Neurons in the middle form a hidden layer, and the rightmost layer is the output layer. In this model, the size of the input layer is equal to the size of the dataset vocabulary. The size of the hidden layer is a tunable hyperparameter, and the output layer is

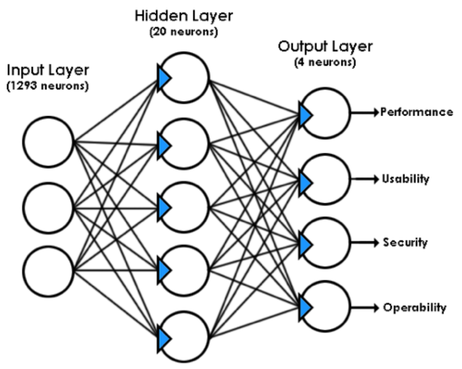


Fig. 1. The ANN model in Our Approach

determined by the number of classes in the dataset being tested. Our ANN model is inspired by the text classifier of a chat-bot [12]. Our application of the ANN to the dataset involved the following steps:

1) *Step 1: Data Preprocessing*

Similar to most machine learning (ML) and big data projects, data preprocessing is a first, necessary step. Software requirements data in natural language usually contains “noise,” (i.e., stop words). Stop words are those words, numbers, special symbols, and individual characters that are frequently seen in text but do not carry significant meanings, such as “the” or “and.” Removing stop words is an essential step in the classification [13]. We implemented a tool that removes stop words and formats the requirements data into a structure that is readable by our models, to make sure we only keep semantically relevant information. Next, our models utilize the bag-of-words approach to tokenize the dataset. Because machine learning techniques cannot directly understand and compute text, the bag-of-words approach helps convert text into vectors of numbers [14]. Specifically, by extending the Natural Language Toolkit (NLTK) library [15], our tool tokenizes each data entry into words. Then, our tool changes each word into lowercase so that same words with upper and lowercases are treated in the same way. For example, “Book” becomes “book” after the transform. For the similar purpose, after that, each word is stemmed by the tool. For instance, “books” and “book” should be considered as the same. Then, each word is added to a list representing the sum vocabulary of the dataset. Finally, each tokenized sentence is then converted into a one-hot NumPy array representation based on the size of the dictionary (0’s) and the words present in the sentence (1’s).

2) *Step 2: Model Training*

The model training process begins with the random initialization of the hidden and output layer’s weights with a mean of 0 (i.e., selecting from [0.0, 2.0) and subtracting 1). After initialization, input can begin to be fed to the model. There are two passes that occur across the network for each training data entry. The first is the feed forward pass (i.e., input travels from the input layer to the hidden layer and then to the output layer). The feed forward process consists of performing a matrix multiplication between the input and hidden layers, and passing the dot product to a sigmoid activation function (i.e., maps values to the range [0,1]) to make the values being learned linearly separable, the results of which are passed to the output layer where the same process is repeated (i.e., from matrix multiplication to the sigmoid activation function). The results produced by the output layer are then used to classify the input, with the node containing the highest score being selected. Since our training is supervised, we can determine the model's level of error by how much our model's prediction missed the correct answers goal of 1 given a range of 0 to 1. The second pass consists of backpropagation through the network (i.e., moving back from the output layer to the hidden layer to correct the weights of neurons based on the level of error). Using the gradient descent method, we can use the sigmoid functions derivative to determine in which direction to adjust the weights to reduce the measured error and improve the fit of the model to the data. Once the weights have been updated, the next feed forward iteration can begin, and so on until the model has trained for the specified number of training epochs.

3) *Step 3: Tuning Hyperparameters*

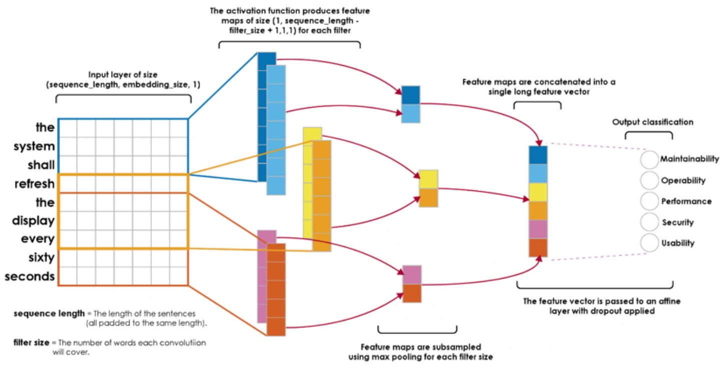
The ANN model is straightforward with only one hidden layer. However, in more complex deep learning models, selecting a higher number of hidden layers may be included in the models tuning. In addition to the size of the hidden layer, we need to tune the remaining hyperparameters used in the model to make the model work effectively. These hyperparameters include:

- *alpha* – the gradient descent parameter that affects the weight adjustment size during backpropagation
- *training epochs* – the number of cycles that the complete set of training data is used to train the model
- *dropout threshold* – a percentage for dropout to prevent overfitting of the model to the dataset by turning off a percentage of neurons during each iteration in order to force the model to learn more generally useful weights

The hyperparameter tuning process is based on noted increases in the models' evaluation metric scores. A hyperparameter baseline is used while each hyperparameter is tested over a series of values resulting in the selection of the highest scoring values.

C. *Convolutional Neural Network (CNN) Model*

Similar to ANNs, CNNs also contain fully-connected layers. CNNs have been widely applied in image related processing. Using TensorFlow [16], our research adapts CNNs to the multi-class classification of NFRs. The structure of the CNN model used in our research is illustrated in Figure 2. The input layer consists of a word embedding representation of an NFR sentence of size (*sequence_length*, *embedding_size*, *1*), where *sequence_length* is the number of tokens in the sentence, and *embedding_size* is an assigned hyperparameter value. Convolutions are performed for each filter size over the input producing feature maps of size (1, *sequence_length* – *filter_size* +1, 1, 1) where *filter_size* represents the number of words observed during each convolution. These filters are learned through the training process. The feature maps for each filter size are then pooled and concatenated into a single long feature vector. This feature vector is then input into an affine layer (i.e., a fully connected layer, the same type of structure as the ANN model) for matrix multiplication and classification.



The activation function produces feature maps of size (1, sequence_length - filter_size + 1,1,1) for each filter
Input layer of size (sequence_length, embedding_size, 1)
the
system
shall
refresh
the
display
every
sixty
seconds
Feature maps are concatenated into a single long feature vector
Output classification
Maintainability
Operability
Performance
Security
Usability
sequence length = The length of the sentences (all padded to the same length).
filter size = The number of words each convolutiion will cover.
Feature maps are subsampled using max pooling for each filter size
The feature vector is passed to an affine layer with dropout applied

Fig. 2. The Architecture of Our CNN Model

The model is optimized using TensorFlow's *AdamOptimizer*, which calculates the model's parameter gradients and performs updates to minimize loss during each training step.

1) *Step 1: Data Preprocessing*

Similar to the ANN model, we need to preprocess the data before its fed to the model. The dataset is read and processed into two equally sized lists, one for the classification labels and the other for the NFRs. The labels are mapped to one-hot array representations and stored in a label dictionary. Our tool removes stop words, then lowercases and stems all the text. Using a series of regular expressions, our tool further cleans the text to formalize symbols, white space, and special characters. After tokenization, each sentence is then padded to a uniform max length. Finally, our tool maps the words in each NFR to an ID and stores them in a new array.

2) *Step 2: Model Training*

The first layer maps vocabulary indices from input NFRs into low-dimensional vectors (i.e., word embeddings). These embeddings are initialized randomly and are learned through the training process. Following the embedding layer is a set of narrow (i.e., no sentence padding has been added at the edges to fit the shape of the filters) convolutional layers for each of three filter sizes (i.e., how many words are viewed at once). Using a pre-labeled dataset, the model learns to identify subject specific features (in our case, NFR specific features) using a series of convolutional filters. To start, the model's filters are initialized to random values, which essentially means the model knows nothing about what it's classifying. Since our dataset is pre-labeled, it enables us to determine if the model was correct or incorrect for each of its classifications. This information enables the model to optimize the weights of its convolutional filters, hence learning the filters shape through the training process to minimize the model's error. The result of each filter convolution is passed to a ReLU non-linearity function. After the filter has convolved over the entire embedding, the resulting output is fed into a max-pooling function which produces a vector of the detected features. Once each filter has been pooled, the results of all the filters are combined into a single large feature vector. This feature vector is then used to perform matrix multiplication and classification in the same manner as the ANN model. Lastly, the model performs the gradient updates to the model's parameters to better fit the model.

3) *Step 3: Tuning Hyperparameters*

TABLE I. ANN Hyperparameters Performance Tuning Metrics

Baseline Hyperparameters:				Training Epochs: 10000				Alpha: 0.2			Hidden Neurons: 10	
ANN – Training Epochs (T)												
T =	**Operability**			**Performance**			**Security**			**Usability**		
	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1
10000	0.55	0.58	0.56	0.44	0.35	0.38	0.83	0.86	0.84	0.84	0.79	0.81
30000	0.81	0.78	0.79	0.85	0.79	0.80	0.85	0.84	0.83	0.86	0.77	0.80
50000	**0.83**	**0.80**	**0.80**	0.85	0.81	0.81	**0.88**	**0.87**	**0.87**	0.85	**0.80**	**0.82**
70000	0.82	0.79	0.79	**0.86**	0.76	0.79	0.81	0.82	0.81	0.84	0.79	0.80
90000	0.82	0.78	0.79	0.85	**0.82**	**0.82**	0.85	0.85	0.84	**0.90**	0.78	**0.82**
ANN – Alpha (A)												
A =	**Operability**			**Performance**			**Security**			**Usability**		
	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1
0.1	0.81	**0.81**	**0.80**	**0.86**	0.80	**0.81**	0.82	0.82	0.81	0.86	0.76	0.80
0.2	0.80	**0.81**	0.79	**0.86**	0.78	**0.81**	**0.86**	**0.86**	**0.85**	**0.88**	0.78	0.82
0.25	**0.84**	0.80	**0.80**	0.85	**0.82**	**0.81**	0.84	0.83	0.83	0.87	**0.81**	**0.84**
0.275	0.58	0.59	0.57	0.35	0.28	0.31	0.62	0.60	0.61	0.84	0.76	0.79
0.3	0.56	0.54	0.54	0.45	0.43	0.43	0.67	0.67	0.67	0.78	0.75	0.76
ANN – Hidden Neurons (H)												
H =	**Operability**			**Performance**			**Security**			**Usability**		
	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1
5	0.81	**0.81**	**0.81**	0.83	**0.82**	**0.81**	**0.85**	**0.85**	0.84	0.88	**0.80**	**0.83**
10	0.81	0.78	0.78	**0.85**	0.78	0.79	0.84	0.84	0.84	0.87	0.78	0.81
15	**0.82**	0.79	0.79	0.66	0.64	0.64	**0.85**	**0.85**	**0.85**	**0.89**	**0.80**	**0.83**
20	0.64	0.66	0.65	0.64	0.59	0.60	0.71	0.64	0.67	0.67	0.65	0.65
25	0.67	0.65	0.66	0.71	0.64	0.66	0.34	0.32	0.33	0.69	0.67	0.67

In addition to filter size, several hyperparameters were tuned in the CNN model to produce more effective results. The hyperparameter tuning process is the same as that described in section II.B. These hyperparameters are listed as follows:

- *training epochs* – the number of iterations through all of the training data
- *training batch size* – the number of training samples per batch
- *embedding dimension* – how much information is stored in each word vector
- *L2 regularization lambda* – used to help prevent the generation of an overly complex model
- *model evaluation frequency* – determines how many steps are taken before the model is evaluated
- *dropout probability* – the percentage of neurons that are turned off during a training step

III. Experimental Evaluation

This section presents our experimental evaluation of the neural network models.

A. Datasets

For the evaluation of the two developed models, we used two largest publicly available labeled NFR datasets: the International Requirements Engineering Conference's 2017 Data Challenge dataset [9] and the Predictors Models in Software Engineering (PROMISE) dataset [17]. We combined these two to obtain a larger and more comprehensive dataset. Overall, the combined dataset include 1,165 unique NFRs falling in 10 classes. Every NFR in the dataset is pre-labeled with its corresponding class. Five classes (Look and Feel, Availability, Fault Tolerance, Scalability, and Legal and Licensing) have very few NFRs in the combined dataset. In order to maintain an overall balanced dataset across all the classes, we trimmed our dataset by dropping the NFRs in these five classes. After that, our NFR dataset consists of 914 NFRs in 5 classes, including maintainability (137), operability (153), performance (113), security (354), and usability (157).

B. Experiment Design

Similar to most machine learning research projects, the dataset is split into three sets: a training set, a validation set, and a testing set [18]. The models see and learn from the training set and fit their weights to the data through the training process. The validation set is used to tune the models parameters, so the models are fed with data in the validation set but are not trained on it. Since the parameters are modified in response to the validation set, it is considered to indirectly affect the model. In order to avoid any biased scores that may be produced from the validation set, a third testing set is used after the model has been trained and validated. The models have never seen the testing set and the models are no longer being modified so the results will not be used to tune the models removing any potential bias.

In our experiment, we perform a stratified (i.e., a proportionate allocation of each class per fold) 10-fold cross-validation of the model's performance. To ensure our experiment is comprehensive and unaffected by any unforeseen conditions, for each model and each tested value of the hyperparameters, we conduct the experiment using 10-fold cross-validation. Then, we obtain results and measure precision, recall, and the F_1 score to compare the results.

C. Experiment Results

While our dataset contained 5 classes, the ANN was trained on only 4, and the number of samples in the security class was reduced to half (177 from 354) due to model stability issues related to the size of the dataset. The CNN was trained on the entire dataset, including all 5 classes and all 354 security samples. Tables I, II, III, and IV list all of the results obtained through our experiment. Each row of results in Tables I, II, III and IV represents the mean average of a 10-fold cross validated set of tests. Tables I and III contain the results obtained during our hyperparameter tuning for each model. For every hyperparameter, we highlight the highest performance metrics values for each class in Tables I and III.

Based on these highest values from 10-fold cross validation, we identify the highest performing hyperparameter settings for each model (see Table V). Using the highest performing hyperparameter settings in Table V, we conduct another round evaluation using 10-fold cross validation. To eliminate any potential bias and minimize any possible impact, we conduct five independent test evaluations. Table II and IV present the results each model can achieve using the highest performing hyperparameter settings. In the best independent iteration (Test #3 for both models), the ANN model scored precision between 82% and 90%, recall between 78% and 85%, with the highest F-score at 84%. The CNN model achieved precision between 82% and 94%, recall between 76% and 97%, with the highest F-score at 92%. Considering the two models, the CNN model classifies one more category than the ANN model. The CNN model's F-score surpassed the ANN model's F-score by 10% for Performance and 9% for Security. The lower number of security samples available to the ANN model may have contributed to its lower score, however, both models were exposed to the same number of Performance samples of which the CNN model still scored a 10% higher F-score. The performance metrics displayed in Table VI are the mean averages of precision, recall, and F-score for each best performing ANN and CNN model highlighted in Tables II and IV. Overall, the ANN model achieved an average F-score at 82%, and the CNN model achieved an average F-score at 86%.

TABLE II. ANN Performance Metrics for the Highest Performing Hyperparameter Settings with 10-Fold Cross-Validation

Test	Operability			Performance			Security			Usability		
#	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1
1	0.82	0.79	0.79	0.85	0.78	0.80	0.84	0.82	0.82	0.83	**0.81**	0.81
2	0.75	0.79	0.76	0.85	0.80	0.80	0.85	0.84	**0.84**	**0.90**	0.75	0.80
3	**0.85**	**0.81**	**0.82**	0.86	0.79	0.81	0.82	**0.85**	0.83	**0.90**	0.78	**0.83**
4	0.76	0.74	0.74	0.85	**0.83**	**0.83**	0.83	0.83	0.82	0.67	0.67	0.67
5	0.79	0.79	0.79	**0.88**	0.80	0.82	**0.86**	0.83	**0.84**	0.86	0.80	**0.83**

TABLE III. CNN Hyperparameters Performance Tuning Metrics

Baseline Hyperparameters:	Training Epochs: 150	Batch Size: 50	Number of Filters: 64	Embedding Dimension: 100	Steps before Evaluation: 100	Dropout: 0.5

CNN – Training Epochs (T)

T =	Maintainability			Operability			Performance			Security			Usability		
	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1
50	**0.92**	0.50	0.65	0.75	**0.74**	0.74	0.92	0.53	0.67	0.69	0.97	0.81	0.76	0.62	0.68
100	0.76	0.52	0.62	0.73	0.66	0.70	**0.95**	0.69	0.80	0.78	**0.99**	0.87	0.79	**0.76**	0.77
150	0.66	0.43	0.52	0.65	0.63	0.64	0.92	0.74	0.82	0.77	0.98	0.86	**0.88**	0.75	**0.81**
200	0.75	**0.71**	**0.73**	0.76	0.61	0.68	0.86	0.68	0.76	0.80	0.96	0.87	0.77	0.74	0.76
250	0.73	0.60	0.66	**0.82**	**0.74**	**0.78**	0.84	**0.81**	**0.83**	**0.82**	0.97	**0.88**	0.81	0.70	0.75

CNN – Batch Size (B)

B =	Maintainability			Operability			Performance			Security			Usability		
	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1
20	**0.85**	**0.71**	**0.77**	**0.85**	0.77	**0.81**	0.75	**0.96**	**0.84**	0.82	0.92	0.87	**0.89**	0.67	0.76
30	0.81	0.69	0.74	0.75	**0.83**	0.79	0.92	0.78	**0.84**	**0.88**	0.95	**0.91**	0.80	0.74	0.77
50	0.80	0.66	0.72	0.80	0.79	0.79	0.93	0.71	0.80	0.82	0.96	0.89	0.83	**0.79**	**0.81**
70	0.76	0.60	0.67	0.72	0.67	0.70	**0.96**	0.59	0.73	0.75	**0.97**	0.85	0.83	0.74	0.79
80	0.82	0.51	0.63	0.72	0.68	0.70	0.84	0.80	0.82	0.75	0.93	0.83	0.80	0.70	0.75

CNN – Number of Filters (N)

N =	Maintainability			Operability			Performance			Security			Usability		
	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1
16	0.80	**0.69**	**0.74**	0.81	0.69	0.74	**0.98**	0.73	0.84	0.74	0.96	**0.94**	0.81	0.63	0.71
32	0.79	0.59	0.68	**0.92**	**0.79**	**0.85**	0.97	0.78	**0.87**	**0.81**	**0.97**	0.88	0.75	**0.78**	0.76
64	0.87	0.61	0.72	0.80	0.67	0.73	0.85	0.74	0.79	0.74	0.95	0.83	**0.89**	**0.78**	**0.83**
128	0.75	0.66	0.70	0.75	0.75	0.75	0.87	**0.79**	0.82	0.80	0.94	0.86	0.79	0.62	0.69
256	**0.89**	0.55	0.68	0.67	0.75	0.71	0.81	0.72	0.76	0.76	0.93	0.84	0.75	0.57	0.65

CNN – Embedding Dimension (E)

E =	Maintainability			Operability			Performance			Security			Usability		
	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1
50	0.80	0.61	0.69	**0.87**	**0.81**	**0.84**	**0.95**	0.59	0.73	0.81	0.96	**0.88**	0.72	**0.81**	0.76
100	0.79	**0.66**	0.72	0.76	0.79	0.78	0.88	0.84	0.86	**0.82**	0.94	**0.88**	0.81	0.65	0.72
150	0.83	**0.66**	**0.74**	0.83	0.75	0.78	0.90	**0.91**	**0.90**	0.79	0.93	0.86	**0.87**	0.75	**0.80**
200	**0.86**	0.61	0.71	0.80	0.76	0.78	0.94	0.69	0.80	0.76	**0.97**	0.85	0.79	0.70	0.74
250	0.74	0.49	0.59	0.84	0.80	0.82	0.91	0.78	0.84	0.74	**0.97**	0.84	0.84	0.65	0.73
300	**0.86**	0.55	0.67	0.67	0.63	0.65	0.88	0.76	0.82	0.77	0.96	0.85	0.71	0.64	0.67

IV. Related Work

Research has attempted to automate the process of analyzing software requirements with different techniques. Cleland-Huang et al. [19] developed NFR-classifier, an algorithm that can first mine and weight indicator terms, and then classify NFR by computing a probability value of its type. Using the Support Vector Machine and Naïve Bayes algorithms, Slankas and Williams [11] developed NFR Locator that examines unstructured documents written in natural language, and then, analyzes, extracts, and classifies NFRs into different categories. Also using the support Vector Machine, Kurtanovic and Maalej [20] developed a supervised learning approach based on meta-data, lexical, and syntactical features. Their approach achieved the highest precision at 92% and the highest recall at 90%. Abad et al. [21] designed a NFR preprocessing approach and investigated 3 types of machine learning algorithms used for classifying NFRs, i.e., topic modeling, clustering, and Naïve Bayes algorithms. Their results indicated that Binarized Naïve Bayes had the best classification performance. Implemented using Word2Vec and TensorFlow, the CNN requirements classifier designed by Dekhtyar and Fong [22] has a similar approach to our CNN model. However, their classifier can only perform binary classification, without the capability of classifying NFRs into their categories.

V. Conclusions

Analyzing NFRs is a very tedious task in the software development lifecycle. However, it is necessary and critical to the subsequent tasks as NFRs directly impact how the software system is developed. If these requirements are not met, the system may become useless. It always costs requirements engineers, software architects, and developers significant effort, costs a large portion of the overall budget, requires strong technical experience, expertise and domain knowledge, and is still error-prone.

In this paper, we present a novel approach of using neural networks to automatically classify NFRs into different categories. Such automatic classification can hugely reduce the cognitive effort required to make sense of NFR statements identified during requirements acquisition. To assess the effectiveness of our approach, we conducted an experimental evaluation using two widely used pre-labeled datasets of software requirements. The experimental results show that our model constructed using CNN can effectively classify NFRs into multiple classes, by achieving precision ranging between 82% and 94%, recall ranging between 76% and 97% with an F-score ranging between 82% and 92%. Our approach is the first and foremost step towards a comprehensive machine learning (ML) system that can help analysts, architects, and developers save unnecessary manual analysis, which can further decrease the possibility of introducing development errors, mistakes, and bugs into the software system. With this comprehensive ML system, the entire software development process will be more efficient, and the overall software quality will be improved. Indirectly, we can increase the customer satisfaction.

The research of applying ML techniques into automatic requirement analysis and classification is still continuing. There are several avenues for future research and improvement. First, we will keep investigating the feasibility and evaluating the effectiveness of other ML techniques in the similar problem domain, such as recurrent neural networks (RNN). Second, even though our approach can automatically classify NFRs into different categories, there does not exist a fully automated, comprehensive, and machine learning-based requirement analysis and software development system yet. We would like to design and build such a system based on our novel approach, to accommodate this need. Third, at the age of big data and ML, many research projects involve datasets at the volume of gigabytes, terabytes, or even petabytes. However, for the research requirement analysis, we were not able to find such a large amount of requirement data, and very few requirement data are labeled, which significantly increases the difficulty of conducting supervised ML research with software requirements. Fourth, we will leverage advanced language models, e.g., BERT [23], to improve the performance of our approach. Fifth, the datasets used in this research have very limited types of NFRs. We will continue searching for larger datasets with more categories of NFRs, so that our approach can be applicable in practice. Therefore, inspired by how human beings learn, we would like to investigate the feasibility of using transfer learning to automatically analyze software requirements.

TABLE IV. CNN Performance Metrics for the Highest Performing Hyperparameter Settings with 10-fold Cross Validation

Test #	Maintainability			Operability			Performance			Security			Usability		
	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1	Precision	Recall	F_1
1	**0.88**	0.50	0.64	0.71	0.74	0.72	0.91	0.80	0.85	0.82	**0.97**	0.89	0.78	0.78	0.78
2	**0.88**	0.72	0.79	0.75	0.84	0.80	0.92	0.84	0.88	**0.88**	0.96	**0.92**	0.82	0.72	0.77
3	0.82	**0.83**	**0.82**	**0.89**	0.76	0.82	**0.94**	**0.89**	**0.91**	**0.88**	**0.97**	**0.92**	**0.86**	**0.80**	**0.83**
4	0.74	0.35	0.47	0.78	**0.91**	**0.84**	0.77	0.88	0.82	0.83	0.96	0.89	0.79	0.64	0.71
5	0.75	0.58	0.66	0.67	0.71	0.69	0.74	0.69	0.71	0.83	0.90	0.86	0.80	0.79	0.80

TABLE V. Highest Performing Hyperparameter Settings Based on F-score

ANN	
Training Epochs	50,000
Alpha	0.25
Hidden Neurons	5
CNN	
Training Epochs	250
Batch Size	20
Number of Filters	32
Embedding Dimension	150

Unlike the condition that, in traditional ML projects, training data and testing data come from the same domain, share the same format, and have the same input feature space and distribution, transfer learning can train a model with the data in one domain, then be applied to another related domain [24]. It is especially attractive when the training data are not enough or inaccessible. It is very similar to how human beings learn knowledge, i.e., people read textbooks, listen to audios, and watch videos, then understand knowledge. For example, once they learn what computer security is, by reading books, papers, and webpages, next time when they read a new sentence or a new paragraph in software requirements, the knowledge they learn can remind them to possibly relate the text to security. Thus, the hypothesis is that we can construct a transfer learning model that is able to be trained with a large number of different categories of text data, then automatically classify software requirements.

ACKNOWLEDGMENT

This project was partially supported by the National Science Foundation under NSF 1663184. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPU used for this research. We would like to thank Google Cloud for supporting our research.

REFERENCES

[1] I. Sommerville, *Software engineering*. Pearson, 2016.

[2] H. Krasner, "The Cost of Poor Quality Software in the US: A 2018 Report," 2018.

[3] R. B. Svensson, T. Gorschek, and B. Regnell, "Quality Requirements in Practice : An Interview Study in Requirements Engineering for," *Requir. Eng. Found. Softw. Qual. Proc. 15th Int. Work. Conf. REFSQ 2009*, pp. 218–232, 2009.

[4] A. Borg, A. Yong, P. Carlshamre, and K. Sandahl, "The Bad Conscience of Requirements Engineering : An Investigation in Real-World Treatment of Non-Functional Requirements," *Third Conf. Softw. Eng. Res. Pract. Sweden (SERPS'03), Lund*, pp. 1–8, 2003.

[5] D. Ameller, C. Ayala, J. Cabot, and X. Franch, "How do software architects consider non-functional requirements: An exploratory study," *2012 20th IEEE Int. Requir. Eng. Conf. RE 2012 - Proc.*, pp. 41–50, 2012.

[6] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*. Boston, MA: Springer US, 2000.

[7] C´ and ıcero N. dos Santos, "Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts," *25th Int. Conf. Comput. Linguist.*, pp. 69–78, 2014.

[8] G. Hinton *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, Nov. 2012.

[9] 25th IEEE International Requirements Engineering Conference, "RE'17 Data Challenge." [Online]. Available: http://ctp.di.fct.unl.pt/RE2017//pages/submission/data_papers/. [Accessed: 21-Jan-2019].

[10] Roman, "A taxonomy of current issues in requirements engineering," *Computer (Long. Beach. Calif).*, vol. 18, no. 4, pp. 14–23, Apr. 1985.

[11] J. Slankas and L. Williams, "Automated extraction of non-functional requirements in available documentation," in *2013 1st International Workshop on Natural Language Analysis in Software Engineering, NaturaLiSE 2013 - Proceedings*, 2013, pp. 9–16.

[12] "Text Classification using Neural Networks – Machine Learnings." [Online]. Available: https://machinelearnings.co/text-classification-using-neural-networks-f5cd7b8765c6. [Accessed: 12-Feb-2019].

[13] A. Rajaraman and J. D. Ullman, "Mining of Massive Datasets: Data Mining (Ch01)," *Min. Massive Datasets*, vol. 18 Suppl, pp. 114–142, 2011.

[14] Y. Goldberg and G. Hirst, *Neural Network Methods in Natural Language Processing*. 2017.

[15] "Natural Language Toolkit — NLTK 3.4 documentation." [Online]. Available: https://www.nltk.org/. [Accessed: 29-Jan-2019].

[16] "TensorFlow." [Online]. Available: https://www.tensorflow.org/. [Accessed: 29-Jan-2019].

[17] T. Menzies, B. Caglayan, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan, "The PROMISE repository of empirical software engineering data." June, 2012.

[18] B. D. Ripley, *Pattern recognition and neural networks*. Cambridge University Press, 1996.

[19] J. Cleland-Huang, R. Settimi, Xuchang Zou, and P. Solc, "The Detection and Classification of Non-Functional Requirements with Application to Early Aspects," in *14th IEEE International Requirements Engineering Conference (RE'06)*, 2006, pp. 39–48.

[20] Z. Kurtanovic and W. Maalej, "Automatically Classifying Functional and Non-functional Requirements Using Supervised Machine Learning," *Proc. - 2017 IEEE 25th Int. Requir. Eng. Conf. RE 2017*, pp. 490–495, 2017.

[21] Z. S. H. Abad, O. Karras, P. Ghazi, M. Glinz, G. Ruhe, and K. Schneider, "What Works Better? A Study of Classifying Requirements," *Proc. - 2017 IEEE 25th Int. Requir. Eng. Conf. RE 2017*, no. 1, pp. 496–501, 2017.

[22] A. Dekhtyar and V. Fong, "RE Data Challenge: Requirements Identification with Word2Vec and TensorFlow," *Proc. - 2017 IEEE 25th Int. Requir. Eng. Conf. RE 2017*, pp. 484–489, 2017.

[23] J. Devlin, M.-W. Chang, K. Lee, K. T. Google, and A. I. Language, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding."

[24] K. Weiss, T. M. Khoshgoftaar, and D. D. Wang, *A survey of transfer learning*, vol. 3, no. 1. Springer International Publishing, 2016.