2017 IEEE 25th International Requirements Engineering Conference

RE Data Challenge: Requirements Identification with Word2Vec and TensorFlow

Alex Dekhtyar
Department of Computer Science and Software Engineering
California Polytechnic State University
San Luis Obispo, USA
dekhtyar@calpoly.edu

Vivian Fong
Department of Computer Science and Software Engineering
California Polytechnic State University
San Luis Obispo, USA
vfong01@calpoly.edu

***Abstract*—Since their introduction over a year ago, Google's TensorFlow package for learning with multilayer neural networks and their Word2Vec representation of words have both gained a high degree of notoriety. This paper considers the application of TensorFlow-guided learning and Word2Vec-based representations to the problems of classification in requirements documents. In this paper, we compare three categories of machine learning techniques for requirements identification for the SecReq and NFR datasets. The first category is the baseline method used in prior work: Naïve Bayes over word count and TF-IDF representations of requirements. The remaining two categories of techniques are the training of TensorFlow's convolutional neural networks on random and pre-trained Word2Vec embeddings of the words found in the requirements. This paper reports on the experiments we conducted and the accuracy results we achieved.**

I. Introduction

Using text classification and information retrieval (IR) on software requirements analysis presents a number of challenges that are not always observed when similar techniques are applied to different domains.

Compared to the traditional size of individual documents, single requirements documents are very short. Individual requirements documents come from different software projects and are written using disparate technical vocabulary. Consequently, learning robust and accurate predictive models over sets of requirements and applying models learned from analyzing requirements from one project to the analysis of another is very difficult.

Recently, Mikolov et al. have conducted an extensive study of word embeddings: a dense representation of words using multidimensional vectors of relatively low dimensionality [1]. Their work introduced a word embedding model called Word2Vec which produces such embeddings by analyzing word co-occurrence in a very large corpus (Google News corpus) and optimizing the probability of predicting, given an embedding of a single word, the context in which this word occurs. In a separate but related development, Google has released TensorFlow, a library for training multilayer neural networks, which form the basis for Google's research in the area of Deep Learning.

The use of the Word2Vec model and convolutional neural networks (CNNs) has clear appeal in the context of requirements analysis. Word2Vec representations of individual words are constructed using a very large corpus of underlying text – something that the analysis of a single requirements document cannot offer. By using Word2Vec word embeddings, we may be able to overcome the challenge posed by the small size of the corpus. CNNs, in turn, provide a highly customizable family of learners that can be used efficiently to learn relationships between requirements.

This paper undertakes the initial study of the use of Word2Vec and TensorFlow for software requirements analysis. As our starting point, we elected the RE Data Challenge Area 2 dubbed "Requirements Identification". For both datasets provided, we compared three classifiers: Naïve Bayes, which serves as our baseline, plus two TensorFlow-based classifiers, one of which uses random assignment of word embeddings to initialize the learner while the other uses pre-trained Word2Vec embeddings. These two methods allow us to observe the efficacy of using CNNs for classification of software requirements and to study the impact of Word2Vec. Section II describes prior work with the two datasets we study in this paper and discusses Word2Vec and TensorFlow. Sections III and IV describe our experiment design and implementation. Finally, Section V details and analyzes the results of our experiments.

II. Background & Related Work

The RE Data Challenge Area 2 focuses on the task of requirements identification, specifically on detecting security-related requirements (SecReq dataset) and distinguishing functional and non-functional requirements (NFR dataset).

A. Datasets

The SecReq[1] dataset was assembled to help research in improving the task of security requirements elicitation

[1]http://www.se.uni-hannover.de/pages/en:projekte_re_secreq

2332-6441/17 $31.00 © 2017 IEEE
DOI 10.1109/RE.2017.26

IEEE computer society

[2]. The dataset consists of requirements labeled as either security-related or not security-related. Knauss et al. trained a Naïve Bayes classifier on the data [3], and by mixing requirements from different projects and running cross-validation, they were able to achieve 78% precision, 91% recall, and 84% F_1-score (and consequently, 88% F_2-score) [3]. In our experiment, we attempt to reproduce these results with Naïve Bayes, and we treat these results as our baseline.

The Quality Attributes (NFR)[2] dataset is comprised of labeled non-functional requirements (NFRs) and functional requirements (FRs) written for a software product by the Department of Homeland Security. The NFRs also carry an additional quality attribute label (i.e. scalability, performance, look and feel). Cleland-Huang et al. focused on the task of classifying NFR quality attributes [4]. They proposed an IR approach of building an explicit supervised learning-like model with pre-labeled training sets and a manual feature extraction process. The classifier produced between 60% and 90% recall for the various NFR quality attributes [4]. We instead concentrate on the binary classification task of distinguishing FRs and NFRs to mirror the task proposed for the SecReq dataset, leaving the problems studied by Cleland-Huang et al. [4] to future work.

B. Word2Vec

Word2Vec[3], introduced by Mikolov et al. at Google, is a collection of shallow neural network models that are designed to learn "high-quality distributed vector representations that capture a large number of precise syntactic and semantic word relationships" [1], [5]. These word vector representations, also known as *word embeddings*, can be of several hundred dimensions. Word embeddings can be combined to represent a document.

Word2Vec's novelty lies in its ability to convey semantic relationships between words; it can determine that *'France'* is to *'Paris'* as *'Germany'* is to *'Berlin'* [5]. The most interesting property of Word2Vec is its ability to perform vector operations on embeddings to capture linguistic regularities. For example, the vector for *'king'*, minus the vector for *'man'*, plus the vector for *'queen'* resembles the vector for *'woman'*, showcasing the ability to extract complex natural language relationships [5]. Google has released a Word2Vec model pre-trained on the 100 billion word Google News corpus, producing 3 million 300-dimension word embeddings for researchers to use in their work.

C. TensorFlow & Convolutional Neural Networks

TensorFlow is "an open-source software library for numerical computation using data flow graphs"[4]. Mathematical operations represent the graph nodes while multidimensional arrays (called *tensors*) represent the edges in between. Tensors flow through the nodes of the graph and produce some output, hence the library's name. Google developed and released TensorFlow to allow the machine learning community to easily build and train neural networks on a variety of environments [6].

TensorFlow can be used to build convolutional neural networks (CNNs) – multilayer feed-forward neural networks that utilize convolving filters and spatial subsampling to force the extraction of local features [7]. CNNs, unlike standard fully-connected feed-forward networks, take advantage of location and order of inputs when extracting features, making them suitable for tasks like image processing and speech recognition where neighboring variables are important to the makeup of the input [7].

A typical CNN has a number of convolution and subsampling layers plus optional fully-connected layers [7]. In image recognition tasks, the convolutional layer receives a 2D image of pixels with multiple channels as input (e.g. RGB images result in 3 channels). In this layer, numerous filters are made to convolve across the image, capturing all of its subareas. Each convolution of a filter results in a window where the activation function is applied, with added bias, to extract features. The features from each convolution for a filter form a feature map, which is then subsampled by mean or max-pooling. The end product is a feature vector that can optionally pass through fully-connected layers before reducing (with a softmax layer) to the output layer. Figure 1 illustrates the discussed architecture.

Although designed originally for computer vision [7], [8], CNNs perform well on NLP tasks as well [9], [10]. With sentence classification, the CNN can intake a 2D embedding matrix built by concatenating the embeddings representing each word in the sentence. By default, these embeddings are vectors of size d initialized with random values that will serve as a starting point for learning the weights through training. The embedding matrix can also be initialized with pre-trained word embeddings, such as Word2Vec, that can immediately provide meaningful weights to our vocabulary.

In the convolution layers for a sentence classification CNN, a series of convolutions with numerous filters of one or more sizes are made, representing the various n-grams in the sentence. The activation function is then applied to each convolution, creating a feature map for each filter which are then subsampled and assembled into a final feature vector. From there, the feature vector is processed similarly to a standard computer vision CNN. Figure 2 shows an example single channel CNN model for sentence classification.

Kim and Zhang et al. explore a few regularization methods to tackle overfitting [10], [11]. One method is *dropout* where features are randomly set to 0 in the softmax layer [12]. Additionally, a constraint to the l_2-norm (i.e. scaling the l_2-norm if the value exceeds a defined threshold) can also be applied. These configurations plus the hyperparameters needed to configure a CNN are outlined in Table I.

[2]http://openscience.us/repo/requirements/requirements-other/nfr.html
[3]https://word2vec.googlecode.com
[4]https://www.tensorflow.org/

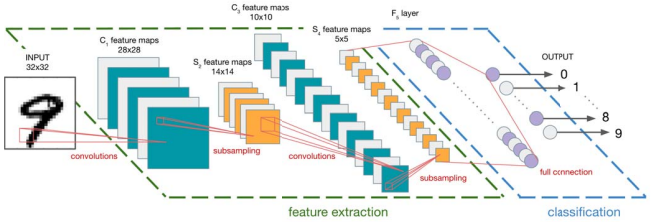

Figure 1: CNN architecture for image classification (adapted from LeCun et al. [7]).

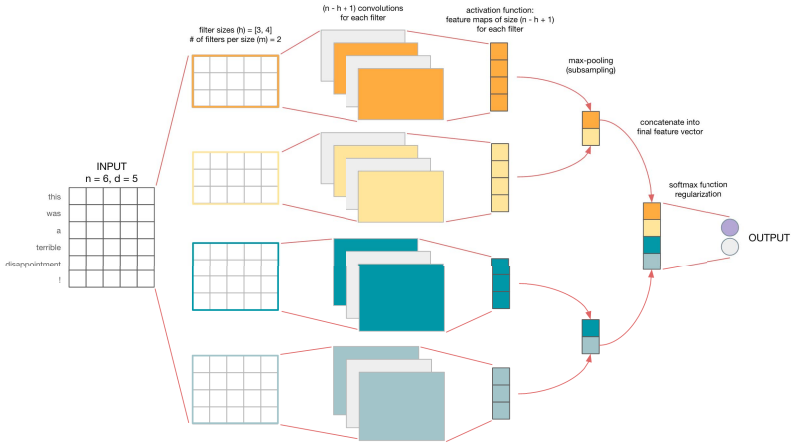

Figure 2: CNN architecture for sentence classification (adapted from Zhang et al. [11]).

Symbol	Description	Symbol	Description
d	embedding dimension	f	activation function
h	filter size(s)	s	subsampling method
m	# of filters per h	p	dropout rate
		l_2	l_2-norm constraint

Table I: Hyperparameters for sentence classifier CNN.

D. Measures

Our results are evaluated using precision, recall, and F_β-score which are calculated by evaluating the true positives (TP), false positives (FP), and false negatives (FN) from the predictions. *Precision* is the percentage of retrieved requirements that are relevant, and *recall* is the percentage of relevant requirements that were successfully retrieved:

$$precision = \frac{TP}{TP + FP} \qquad recall = \frac{TP}{TP + FN}$$

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

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

In our domain, a higher recall is considered more important than higher precision, so in addition to F_1-measure, we consider F_2-measure in this paper.

We also calculate cross-entropy loss for our CNN experiments. In neural networks, cross-entropy loss H measures how close the probability distribution between the true distribution p and the predicted distribution q [13]:

$$H(p, q) = -\sum_x p(x) \log q(x)$$

Minimizing cross-entropy loss thus reduces the error in our predictions.

III. Design

In this section, we describe the design of our experiments for both the SecReq and NFR datasets.

A. Baselines

First, we start off by either reproducing or establishing the baseline for both datasets using Naïve Bayes. Since the SecReq dataset is already designed as a binary classification task, the results from a Naïve Bayes approach have already been documented [3]. Thus, for SecReq we simply follow the general procedures conducted by Knauss et al. to reproduce their results.

The NFR dataset however has had more in-depth exploration on the multi-label classification task of identifying quality attributes in non-functional requirements [4]. To align the NFR dataset with the SecReq experiment, we

simply build a Naïve Bayes classifier to determine our own baseline metrics to identifying functional requirements from non-functional requirements. We plan to address the quality attribute identification task in future work.

For both datasets, we implemented two classifiers with different feature vector schemes: word count and TF-IDF.

B. Experiments

Our goal is to observe the performance of CNNs on these datasets compared to the baselines and measure the impact of pre-trained Word2Vec embeddings on the model. To do this, we implement a CNN following the sentence classification design discussed in Section II-C using the TensorFlow library.

Since this paper serves as an initial investigation of this application, we simplify the experiments by limiting the independent variables adjusted for each experiment to the following: (1) word embeddings (either random or Word2Vec), (2) m, the number of filters per filter size, and (3) the number of epochs the model trains for. As inflating the latter two factors can easily lead to overfitting, our goal is to inspect what combinations produce the best results while minimizing complexity.

The pre-trained Word2Vec embeddings have a dimension d of 300; for random word embeddings, we set d to 128. As for the remaining hyperparameters, we follow Kim's suggestions and set filter sizes h as [3, 4, 5], activation function f as rectified linear unit, subsampling method s as max-pooling, and training dropout rate p as 0.5 [10]. Dropout will not be included during testing. We did not incorporate l_2-norm constraints.

We run the same set of configurations with 10-fold cross validation and batch sizes of 64 for both datasets. Afterwards, we collect the metrics described in Section II-D.

IV. Implementation

A. Naïve Bayes

For our baselines, we built simple Naïve Bayes classifiers using Scikit-Learn[5], a well-established open-source machine learning library in Python. We used their `MultinomialNB` version of Naïve Bayes with both `CountVectorizer` and `TfidfVectorizer` as feature extractors. The vectorizer utility classes help conduct case normalization and stop word removal. We used Scikit-Learn's built-in K-fold cross-validation.

B. Word2Vec & Gensim

We downloaded the binary for the Google News pre-trained Word2Vec model off the Word2Vec website[6]. Gensim[7], a Python framework for vector space modeling, is equipped with APIs for handling Word2Vec models, making it convenient for us to load and use the pre-trained models in our system.

C. TensorFlow

This section simply outlines the APIs used to help realize our CNN model design.

First, we chose to clean the data as follows: we perform case normalization, stop word removal, as well as separate contractions (i.e. *"don't"* becomes *"do"* and *"n't"*), and add spaces around punctuation. Afterwards, with the help of TensorFlow's vocabulary processor we make a vocabulary index mapping each word in the corpus to a number, and then convert all of our documents into vocabulary index vectors. These become the inputs to the embedding layer.

The *embedding layer* is where we construct the embedding matrix to represent our input document. To do this, a look-up table needs to be made for the entire corpus vocabulary, mapping each vocabulary word index to its embedding. These embeddings (each of size d) are either randomly initialized or are taken from the Gensim-loaded Word2Vec model. If a word is not represented in the Word2Vec model, its embedding is initialized as a zero vector. (We will need to investigate better methods of initializing embeddings for these unrepresented words in our future work.) TensorFlow's `embedding_lookup` operation transforms the input vocabulary index vector of size n into a 2D embedding matrix of shape $n \times d$.

Following the embedding layer is a series of *convolution and max-pooling layers*, one set for each filter size h. For each h, we create m filters of shape $h \times d$. The following steps describe the operations for just one filter size. The `conv2d` function takes in the embedding matrix and a tensor of the filter shape to create $n-h+1$ convolutions for each of the m filters. All $m \cdot (n-h+1)$ convolutions are then added the bias value b and are funneled into the nonlinearity activation function f. In our case, f is the rectified linear units (ReLU) function. The result from this is a feature map $\mathbf{c}$ of size $n-h+1$ for each filter.

Once the feature maps are produced, the features are then subsampled with max-pooling. TensorFlow's `max_pool` function takes in the feature maps $\mathbf{c}$ and outputs each max feature $\hat{c}$ concatenated into a tensor $\mathbf{z}$. The pooled features for each convolutional/max-pooling layers are concatenated and flattened into one long tensor and fed into the dropout layer, where, in training, the pooled features are switched off with probability p.

Last, the features that made it through the dropout layer reach the *output layer*. To score the labels, matrix multiplication is performed on the final feature vector to transform it into a vector with correspondence scores for each class. The class with the highest score becomes the prediction for that input.

[5] http://scikit-learn.org/
[6] word2vec.googlecode.com
[7] https://radimrehurek.com/gensim/

V. Results & Discussion

Tables II and III showcase CNN experimental results with various word embedding, filter, and epoch configurations and compare them with our Naïve Bayes baselines.

A. SecReq

For our SecReq baseline, we managed to closely reproduce the Naïve Bayes metrics from Knauss et al. [3]. As shown at the bottom of Table II, our results with word count vectors fall just short under the metrics from the original paper. This can likely be attributed to rounding errors as well as slight differences in implementation (we utilized Scikit-Learn's `MultinomialNB` class whereas Knauss et al. built the classifier from scratch), features (we used word counts instead of word presence), and preprocessing steps (the original paper did not detail whether the documents were cleaned). Compared to word count vectors, our TF-IDF implementation scored 5.43% higher in precision but lost a considerable 18.25% in recall.

Our CNN results show that for random word embeddings, we can easily improve precision, with the highest score at 87.85% with 50 filters and 100 epochs. Recall, on the other hand, did not come close to the word count baseline, resulting in poor F_1 and F_2 scores.

The addition of Word2Vec on the other hand provided an overall boost in scores and helped us reach our goal: with 30 filters and 100 epochs, we scored an F_1-score of 91.34% and F_2-score of 91.33%. This configuration allowed precision to nearly match recall, up 13.5% compared to the baseline.

B. NFR

For the NFR dataset, we established our own Naïve Bayes baseline in a way similar to our SecReq baseline. Both word count and TF-IDF variations performed comparably, with word count vectors scoring a F_2-score of 86.83%. Our CNN experiments easily outperformed the baseline, both with random embeddings and Word2Vec. Word2Vec again contributed toward a drastic boost in performance, achieving an F_2-score of 91.96% with 50 filters and 100 epochs. 30 filters and 60-80 epochs fell a close second, suggesting that the variation of filters between 30 and 50 might not have great significance in performance.

C. Analysis

Overall, the results show that applying CNNs to this domain produce promising results. The model tended to raise both precision and recall together, thus significantly improving precision in the SecReq dataset while keeping recall at the same level as the baseline. More importantly, the addition of pre-trained Word2Vec word embeddings reliably improved the performance of our model for both datasets.

VI. Conclusions & Future Work

In this paper, we presented our initial study on the usefulness of Word2Vec word embeddings and convolutional neural networks for analysis of requirements documents. In reflecting on the results, we acknowledge that our initial study included two well-studied requirements datasets on which Naïve Bayes classifiers already perform well on. Having said that, our CNN classifier equipped with Word2Vec embeddings provided an F_1-score lift of over 7% over baseline methods for the SecReq dataset and around 5.5% for the NFR dataset. More importantly, this approach provided a significant boost to precision in each case without sacrificing recall.

We therefore reach two preliminary conclusions. First, we observe that CNN classifiers can be successfully trained on relatively small collections of requirements documents to recognize a variety of requirements properties. Second, we see that the Word2Vec embeddings, when used to represent individual words in the requirements, provide a real and measurable boost to the classification accuracy.

To be able to strengthen these conclusions, we plan to conduct additional studies of requirements documents using Word2Vec and TensorFlow. For our next set of experiments we plan to classify the non-functional requirements in the NFR dataset into the quality attribute categories studied by Cleland-Huang et al. [4] (scalability, performance, etc.). We plan to build classifiers using CNNs and Word2Vec for additional requirements datasets. We also plan to study the use of Word2Vec embeddings for requirements representations in automated trace recovery problems.

References

[1] 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*, ser. NIPS'13. Curran Associates Inc., 2013, pp. 3111–3119.

[2] S. H. Houmb, S. Islam, E. Knauss, J. Jürjens, and K. Schneider, "Eliciting security requirements and tracing them to design: An integration of common criteria, heuristics, and UMLsec," *Requir. Eng.*, vol. 15, no. 1, pp. 63–93, Mar. 2010.

[3] E. Knauss, K. Scheider, 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.

[4] J. Cleland-Huang, R. Settimi, X. Zou, and P. Solc, "Automated classification of non-functional requirements," *Requirements Engineering*, vol. 12, no. 2, pp. 103–120, 2007.

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

[6] S. Pichai, "TensorFlow: smarter machine learning, for everyone," *Google Official Blog*, 2015.

Embeddings	# Filters per Size	# Epochs	Loss	Accuracy	Precision	Recall	F1-Score	F2-Score
random	30	20	0.3631	0.8207	0.8332	0.7830	0.7884	0.7820
random	30	40	0.3415	0.8347	0.8457	0.7950	0.8069	0.7974
random	30	60	0.3056	0.8684	0.8665	0.8408	0.8475	0.8422
random	30	80	0.2943	0.8567	0.8607	0.8290	0.8344	0.8292
random	30	100	0.3070	0.8684	0.8631	0.8422	0.8480	0.8436
random	50	20	0.3911	0.8207	0.8261	0.7793	0.7872	0.7798
random	50	40	0.3432	0.8585	0.8541	0.8328	0.8377	0.8336
random	50	60	0.3025	0.8705	0.8764	0.8413	0.8524	0.8444
random	50	80	0.3484	0.8647	0.8643	0.8322	0.8403	0.8340
random	50	100	0.3253	0.8846	0.8785	0.8686	0.8706	0.8687
word2vec	30	20	0.3490	0.8585	0.8580	0.8343	0.8381	0.8342
word2vec	30	40	0.2565	0.8884	0.8815	0.8729	0.8745	0.8729
word2vec	30	60	0.2248	0.8984	0.8928	0.8869	0.8877	0.8867
word2vec	30	80	0.2118	0.9125	0.9018	0.8993	0.8984	0.8985
word2vec	**30**	**100**	**0.2018**	**0.9105**	**0.9152**	**0.9138**	**0.9134**	**0.9133**
word2vec	50	20	0.3224	0.8803	0.8686	0.8681	0.8658	0.8666
word2vec	50	40	0.2393	0.9044	0.9028	0.8980	0.8953	0.8957
word2vec	50	60	0.2092	0.9082	0.9026	0.8957	0.8976	0.8961
word2vec	50	80	0.2061	0.9144	0.9069	0.9032	0.9025	0.9024
word2vec	50	100	0.2095	0.9043	0.9005	0.8903	0.8922	0.8904
Naive Bayes – Knauss et at. [3]				*-*	*0.78*	*0.91*	*0.84*	*0.88*
Naive Bayes – Word Count				*0.8742*	*0.7768*	*0.9046*	*0.8328*	*0.8758*
Naive Bayes – TF-IDF				*0.8403*	*0.8311*	*0.7221*	*0.7626*	*0.7416*

Table II: SecReq Metrics

Embeddings	# Filters per Size	# Epochs	Loss	Accuracy	Precision	Recall	F1-Score	F2-Score
random	30	20	0.3377	0.8631	0.8711	0.8435	0.8491	0.8439
random	30	40	0.3360	0.8711	0.8787	0.8573	0.8619	0.8578
random	30	60	0.3339	0.8840	0.8925	0.8669	0.8745	0.8688
random	30	80	0.3591	0.8937	0.9063	0.8772	0.8836	0.8781
random	30	100	0.3596	0.8785	0.8789	0.8760	0.8758	0.8756
random	50	20	0.3832	0.8471	0.8627	0.8235	0.8321	0.8246
random	50	40	0.3371	0.8745	0.8811	0.8630	0.8662	0.8630
random	50	60	0.3547	0.8857	0.8953	0.8659	0.8733	0.8675
random	50	80	0.3537	0.8825	0.8837	0.8730	0.8758	0.8736
random	50	100	0.3954	0.8679	0.8729	0.8558	0.8588	0.8557
word2vec	30	20	0.3134	0.8905	0.8932	0.8802	0.8842	0.8813
word2vec	30	40	0.2520	0.9114	0.9100	0.9024	0.9052	0.9033
word2vec	30	60	0.2302	0.9211	0.9229	0.9135	0.9170	0.9146
word2vec	30	80	0.2314	0.9226	0.9227	0.9153	0.9172	0.9156
word2vec	30	100	0.2439	0.9005	0.9032	0.8993	0.8989	0.8986
word2vec	50	20	0.2894	0.8987	0.8968	0.8914	0.8922	0.8913
word2vec	50	40	0.2362	0.9132	0.9131	0.9043	0.9075	0.9053
word2vec	50	60	0.2352	0.9100	0.9084	0.9037	0.9044	0.9036
word2vec	50	80	0.2500	0.9194	0.9236	0.9095	0.9137	0.9105
word2vec	**50**	**100**	**0.2408**	**0.9259**	**0.9268**	**0.9187**	**0.9216**	**0.9196**
Naive Bayes – Word Count				*0.8691*	*0.8678*	*0.8707*	*0.8664*	*0.8683*
Naive Bayes – TF-IDF				*0.8654*	*0.8683*	*0.8634*	*0.8634*	*0.8628*

Table III: NFR Metrics

[7] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, vol. 86, no. 11, 1998, pp. 2278–2324.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, pp. 1097–1105.

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

[10] Y. Kim, "Convolutional neural networks for sentence classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014*, 2014, pp. 1746–1751.

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

[12] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012.

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.