Long Term-short Memory Neural Networks and Word2vec for Self-admitted Technical Debt Detection

Rafael Meneses Santos[1], Israel Meneses Santos[2], Methanias Colaço Rodrigues Júnior[2] and Manoel Gomes de Mendonça Neto[1]
[1]*Graduate Program in Computer Science, Federal University of Bahia, Salvador, Brazil*
[2]*Department of Information Systems, Federal University of Sergipe, Itabaiana, Brazil*

Keywords: Mining Software Repositories, Self-admitted Technical Debt, Long Short-term Memory, Neural Networks, Deep Learning, Word Embedding.

Abstract: Context: In software development, new functionalities and bug fixes are required to ensure a better user experience and to preserve software value for a longer period. Sometimes developers need to implement quick changes to meet deadlines rather than a better solution that would take longer. These easy choices, known as Technical Debt, can cause long-term negative impacts because they can bring extra effort to the team in the future. Technical debts must be managed and detected so that the team can evaluate the best way to deal with them and avoid more serious problems. One way to detect technical debts is through source code comments. Developers often insert comments in which they admit that there is a need to improve that part of the code later. This is known as Self-Admitted Technical Debt (SATD). Objective: Evaluate a Long short-term memory (LSTM) neural network model combined with Word2vec for word embedding to identify design and requirement SATDs from comments in source code. Method: We performed a controlled experiment to evaluate the quality of the model compared with two language models from literature and LSTM without word embedding in a labelled dataset. Results: The results showed that the LSTM model with Word2vec have improved in recall and f-measure. The LSTM model without word embedding achieves greater recall, but perform worse in precision and f-measure. Conclusion: Overall, we found that the LSTM model and word2vec can outperform other models.

1 INTRODUCTION

On a day-to-day basis, developers must meet deadlines and ensure that the software is being developed with quality. The evolution or maintenance of software is a fundamental step to guarantee the quality of the product. The software goes through several modifications that can be requested by users as improvement or correction of an error committed during implementation. In some cases, quick fixes and workarounds must be applied in order for one or more features to be delivered on time. When developers opt for these choices, they can make a trade-off between meeting deadlines and software quality. These shortcuts tend to have a negative impact in the long run, so that short-term gains are obtained. This type of choice is called technical debt.

The term "technical debt" was defined by Ward Cunningham. A technical debt is a debt that the development team takes when it chooses to do something that is easy to implement to meet a short-term goal but can have a negative impact in the future (Cunningham, 1992). When technical debts are not managed and corrected, they can have serious long-term consequences, increasing costs during the maintenance (Seaman and Guo, 2011). Although there are cases of technical debt occurring unintentionally, there are situations in which developers admit that they have produced or found a technical debt. This type of technical debt is called self-admitted technical debt (SATD) (Potdar and Shihab, 2014).

Given the need of better ways to deal with technical debt, some work have been done on how to detect and manage technical debt (Guo and Seaman, 2011; Codabux and Williams, 2013; Nord et al, 2012). The studies have proposed ways of detecting technical debts through manual or automatic analysis of project artifacts, mainly source code.

Santos, R., Santos, I., Rodrigues Júnior, M. and Neto, M.
Long Term-short Memory Neural Networks and Word2vec for Self-admitted Technical Debt Detection.
DOI: 10.5220/0009796001570165
In *Proceedings of the 22nd International Conference on Enterprise Information Systems (ICEIS 2020) - Volume 2*, pages 157-165
ISBN: 978-989-758-423-7

Some studies extracted metrics from the code to obtain indications of possible irregularities that may point to technical debt (Marinescu, 2012). Another important contribution was the definition of various types of technical debt. Some of them are: design debt, requirement debt, defect debt, documentation debt and test debt (Alves et al, 2014).

Potdar and Shihab (2014) have shown that technical debt can be found by analyzing comments in source code. In this case, the technical debts found in the comments are SATDs because the developers explicitly indicate that parts of the code needs changes. Comments may indicate that the code is not complete, does not meet the requirements, needs refactoring or even needs to be completely redone. It has also been found that the most common types of SATDs are design and requirement ones (Maldonado and Shihab, 2015).

Detecting technical debts in comments has some advantages over the source code approach (Maldonado, Shihab, and Tsantalis, 2017). Extracting comments is a simple task, which can be done using even a regular expression. When we use source code it is often necessary to assemble complex structures with high computational cost. Also, in cases of detection from code smells, it is necessary to set thresholds for the metrics been used, a problem that is still being researched.

Despite the potential in detecting technical debt in comments, the manual process is problematic. In projects with thousands of comments, it becomes virtually impossible for developers to look into comments and classify whether that comment refers to a type of technical debt or not. In this way, an automatic process for detecting SATDs in comments is necessary.

Some approaches to automatic detect technical debt in comments have been proposed recently. Maldonado, Shihab, and Tsantalis (2017) proposed an approach that uses natural language processing (NLP) and a maximum entropy classifier. Wattanakriengkrai et al (2018) proposed a combination of N-gram IDF and auto-sklearn classifier for detection of SATDs. Both studies obtained good results, and the approach from Wattanakriengkrai et al. presented an improvement over the maximum entropy classifier. In both cases, these work mainly used design and requirement SATDs for training and testing their models, in addition they made available their dataset so that other researchers can evaluate other classifiers.

Currently, deep learning neural networks have shown impressive results in classification tasks such as image recognition, speech, and text classification (Lecun, Bengio, and Hinton, 2015). Long Short-Term Memory (LSTM) neural networks presented better results than traditional techniques in text classification and sentiment analysis (Zhou et al, 2015; Zhou et al, 2016). The ability to capture temporal and sequential information makes Recurrent Neural Networks (RNN) and LSTMs ideal for text classification tasks (Zhou et al, 2015).

For text classification, one of the best ways to process text data is through word embedding (Mikolov et al, 2015). Mikolov et al (2015) propose a method called Word2vec that can transform a great amount of text in numerical vectors. A neural network can use these numerical vector as input instead of other word representation with no contextual information.

Therefore, based on the results of Maldonado, Shihab, and Tsantalis, (2017) and Wattanakriengkrai et al (2018), in this paper we evaluate a LSTM neural network model and Word2vec to identify design and requirement SATDs from comments in source code. First we train a LSTM neural network with and without Word2vec using the dataset made available by Maldonado and Shihab (2015). Then we apply the training model to classify a test set. The validation process was carried out using 10 projects from the dataset through a leave-one-out cross-project validation process. Finally, the results were compared to Maldonado, Shihab, and Tsantalis, (2017) and Wattanakriengkrai et al (2018), to evaluate the performance of the LSTM network.

The results showed that the LSTM model with Word2vec have improved in recall and f-measure in design SATD classification. The LSTM model without word embedding achieve greater recall, but perform worse in precision and f-measure. In requirement classification, Wattanakriengkrai et al (2018) model precision was greater than any LSTM model, however, the f-measure was similar to that of the LSTM with Word2vec, from the statistical significance point of view. This may have occurred because the database is imbalanced, having more comments without SATDs and little amount of training data in the requirement SATDs. Recall may be more important than precision depending on the problem being discussed (Hand and Christen, 2018). Someone may accept a slightly higher rate of false positives to get more true positives, if the trade-off is considered interesting.

The rest of this paper is organized as follows. In Section 2, we discuss works related to LSTM and detection of SATDs. Section 3 presents the evaluation methodology, the dataset, and a introduction to LSTM and word embedding. Then, in Section 4, we discussed the planning and execution of the experiment. The results of the experiment are presented and discussed in Section 5 as well as the threats to validity. Finally, in Section 6, we present the conclusions of the paper and some possible extensions that can be researched in future works.

2 RELATED WORKS

In our work we use an LSTM neural network to classify SATDs in source code comments. There are related papers that talk about both LSTM in text classification tasks and technical debt detection in source code comments, especially self-admitted technical debts.

Zhou et al (2015) propose a combination of LSTM neural networks and Convolutional neural networks (CNN) to perform text classification and sentence representation. CNN is able to extract high-level information from sentences, forming a sequence of phrases representations, and then feeds an LSTM to obtain the representation of the complete sentence. This approach is particularly suitable for text classification because the model can learn local and global representation of the features in the convolutional layer and temporal representation in the LSTM layer.

The detection of SATDs has been the subject of some research using mainly natural language processing. Potdar and Shihab (2014) extracted 62 comment patterns from projects that can indicate SATDs. They found that technical debt exists in 2.4% to 31% of the files. In most cases, the more experienced developers tend to introduce comments in the code that self-admit a technical debt. Finally, their work presented that only 26.3% to 63.5% of SATDs are resolved in the project.

Maldonado, Shihab, and Tsantalis (2017) proposed an approach to detect SATDs using natural language processing (NLP) and a maximum entropy classifier. In this work, only design and requirement SATDs were analyzed because they are the most common and all the researched projects contains this type of SATDs. They build a dataset of manually labelled comments from 10 projects: Ant, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby and SQuirrel SQL. The results show that the approach presented better results compared to the model that uses comments patterns. Words related to sloppy or mediocre code tend to indicate SATD design, whereas comments with words about something incomplete show indications of requirement SATD.

Wattanakriengkrai et al. (2018) also worked with design and requirement SATD. They proposed a model that combines N-gram IDF and the auto-sklearn machine learning library and compared the results with Maldonado, Shihab, and Tsantalis (2017). The results show that they outperformed the previous model, improving the performance over to 20% in the detection of requirement SATD and 64% in design SATD.

Two studies used mining techniques to classify SATDs (Huang et al, 2018; Liu et al, 2018). The first work proposed a model that uses feature selection to find the best features for training and uses these features in a model that combines several classifiers. The second one introduces a plugin for Eclipse to detect SATDs in Java source code comments. From this tool, the developer can use the model integrated to the plugin or another model for the detection of SATDs. The plugin can find, list, highlight and manage technical debts within the project.

Based on the studies of Maldonado, Shihab, and Tsantalis, (2017) and Wattanakriengkrai et al (2018), in our work we propose to evaluate an LSTM model with and without Word2vec and compare with the results obtained by these approaches. We think that an LSTM neural network can achieve better results in this type of classification task, based on previous work reports on text classification and LSTM (Huang et al, 2018; Young et al, 2018).

3 METHODOLOGY

The main objective of this work is to evaluate an LSTM model with Word2vec for detection of design and requirement SATDs in source code comments. The first step of the work is to load and clean a dataset of SATDs so that they can be properly used by the LSTM model. After cleaning the dataset, we trained the Word2vec and the LSTM network. We classify the test set using the trained model. To perform this procedure, a controlled experiment was defined and executed. This experiment is detailed in Section 4.

Experimentation is a task that requires rigorous planning with well-defined steps (Wohlin et al, 2012). We have to elaborate planning, execution and analysis of the data. From this, it is possible to apply a statistical treatment of the data, with hypothesis

Table 1: Total number of comments by type of project and technical debt.

	Defect	Test	Documentation	Design	Requirement	No Technical Debt	**Total**
Ant	13	10	0	95	13	3967	**4098**
ArgoUML	127	44	30	801	411	8039	**9452**
Columba	13	6	16	126	43	6264	**6468**
EMF	8	2	0	78	16	4286	**4390**
Hibernate	52	0	1	355	64	2496	**2968**
JEdit	43	3	0	196	14	10066	**10322**
JFreeChart	9	1	0	184	15	4199	**4408**
JMeter	22	12	3	316	21	7683	**8057**
JRuby	161	6	2	343	21	4275	**4897**
Squirel	24	1	2	209	50	6929	**7215**
Total	**472**	**85**	**54**	**2703**	**757**	**58204**	**62275**

tests, so that it can be replicated by others and produce reliable information.

3.1 Self-admitted Technical Debt Dataset

One of the main contributions of Maldonado and Shihab (2015) was to build and make available a dataset of SATDs so that other researchers analyze and test their models. The dataset was created by extracting comments from 10 open source software projects. The selected projects were: Ant, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby and SQuirrel SQL. The criterion used for this selection was that the projects should be from different application domains and had a large amount of comments that can be used for classification and analysis of technical debt.

After extracting comments from the projects, the researchers labelled each comment manually with some type of technical debt. The classification was made based on the work of Alves et al. (2014), who presented an ontology of terms that can be applied to define types of technical debt. The types defined were: architecture, build, code, defect, design, documentation, infrastructure, people, process, requirement, service, test automation and test debt. Not all types were used during the labelling process because some of them were not found in code comments. They found technical debt comments of the following types: design debt, defect debt, documentation debt, requirement debt and test debt. Table 1 shows the number of SATDs by type and project. This process resulted in the classification of 62275 comments, being 4071 comments of technical debt of different types and 58204 comments that indicate no technical deb.

The most common types of SATDs found were design (2703 comments) and requirement SATDs (472 comments). The two types have some distinctions which are explained and exemplified below.

Self-admitted design debts are characterized by comments that talk about issues in the way that the code was implemented. The comments usually indicate that the code is poorly constructed and that they need modifications to improve its quality. Modifications can be made through refactoring process or even redoing the code from the beginning. Some examples of comments that contain design SATDs are:

1. *check first that it is not already loaded otherwise consecutive runs seems to end into an OutOfMemoryError or it fails when there is a native library to load several times this is far from being perfect but should work in most cases* (from Ant project)*;*
2. *this is wrongly called with a null handle, as a workaround we return an empty collection* (from ArgoUML project).

In the first comment, the developer explains some troublesome code, but that works most of the time. In addition to explaining the code problem, it is obvious to the reader that this must be solved to prevent the program from throwing exceptions. The second example shows a comment stating that code has a workaround to solve a problem. Words like **workaround**, **stupid** and **needed?** are good candidates to identify comments with SATDs design (Maldonado and Shihab, 2015).

Self-admitted requirement debts are found in comments that talk about incomplete code that do not fully meet some requirement. Some examples are:

1. *TODO: Why aren't we throwing an exception here? Returning null results in NPE and no explanation why. (from ArgoUML project);*
2. *Have we reached the reporting boundary?Need to allow for a margin of error, otherwise can miss the slot. Also need to check we've not hit the window already* (from jMeter project)

In both cases, the developers explain in the comments that the code needs to be completed and question specific points that are missing in the code.

3.2 Long Short-term Memory Neural Network

LSTM was initially proposed by Hochreiter and Schmidhuber (1997) to solve the problem of long sequences by a recurring neural network. In the original approach, updating the weights, which is done by a variation of the Backpropagation algorithm, called Backpropagation Through Time, is affected when the level of recurrence is high, which makes it impossible for the network to memorize long sequences.

Its architecture consists of a set of subnets connected repeatedly, called the memory block, located in the hidden layer. This block contains memories cells with auto connections capable of storing the temporal state of the network, in addition to special units, called gates, which are responsible for controlling the flow of information, as shown in Figure 1. Each block consists of one or more memory cells connected with three gates: forget (Eq. 1), input (Eq. 2) e output (Eq. 3). Each gate has a function that allows you to reset, write and read the operations inside the memory block. Each gate uses a sigmoid logistic function (σ) to flatten the values of these vectors between 0 (closed gate) and 1 (open gate).
The forget gate defines whether previous state activation will be used in memory. The input gate defines how much of the new state calculated for the current input will be used, and finally, the output gate defines whether the internal state will be exposed to the rest of the network (external network). Then a hyperbolic tangent layer creates a vector of new candidate value $\tilde{C}_t$ (Eq. 4) that can be added in the cell.

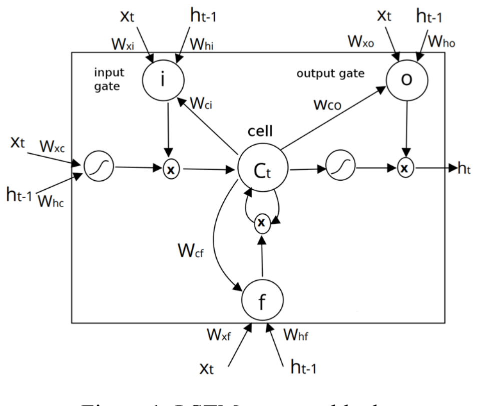

Figure 1: LSTM memory block.

C_t is the internal memory unit, which is a combination of the previous memory C_{t-1} multiplied by the forget gate and the candidate values $\tilde{C}_t$ multiplied by the input gate (Eq. 5). Intuitively, one realizes that memory is a combination of memory in the previous time with the new one in the current time.

$$f_t = \sigma\big(W_{xf} * x_t + W_{hf} * h_{t-1} + W_{cf} * c_{t-1} + b_f\big) \tag{1}$$

$$i_t = \sigma(W_{xi} * x_t + W_{hi} * h_{t-1} + W_{ci} * c_{t-1} + b_i) \tag{2}$$

$$o_t = \sigma(W_{xo} * x_t + W_{ho} * h_{t-1} + W_{co} * c_t + b_o) \tag{3}$$

$$\tilde{C}_t = tanh(W_{xc} * x_t + W_{hc} * h_{t-1} + b_c) \tag{4}$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{5}$$

Given this memory C_t, it is finally possible to calculate the output of the hidden state h_t by multiplying the memory activations with the output gate (Eq. 6).

$$h_t = o_t * tanh(C_t) \tag{6}$$

The variables i_t, f_t, o_t, c_t, and h_t are vectors representing values in time t. W_* are matrices of weight connected to different gates, and b_* are the vectors that correspond to bias.

3.3 Word Embedding

Word Embeddings are methods that transform a sequence of words into a low-dimensional numerical representation. In this way, it is possible to model a language or extract features from it to use as input in machine learning algorithms and other natural language processing activities. One of the most famous methods of word embedding was proposed by Mikolov et al (2015) and is called Word2vec.

Word2vec is a model based on neural networks that can process a large amount of text to and store context information of words. Finally, Word2vec returns a vector space with information for each word. For text classification, we can use Word2vec in the text pre-processing step.

4 EXPERIMENT

We follow a experimental process to evaluate our LSTM model results based on Wohlin's guidelines (Wohlin et al, 2012). In this section, we will discuss planning and execution of the experiment.

4.1 Goal Objective

The objective of this study is to evaluate, through a controlled experiment, the efficiency of the LSTM neural network with Word2vec in design and requirement SATDs classification in source code comments. The experiment was done by using a dataset build by Maldonado and Shihab (2015) to train the LSTM model and we compare the results to those from the studies of Maldonado, Shihab, and Tsantalis, (2017) and Wattanakriengkrai et al, (2018). Wattanakriengkrai et al. (2018) combined N-gram IDF and auto-sklearn, and Maldonado, Shihab, and Tsantalis, (2017) used maximum entropy classifier.

The objective was formalized using the GQM model proposed by Basili and Weiss (1984): **Analyze** the LSTM neural network with Word2vec, **with the purpose of** evaluating it (against results of algorithms evaluated in previous works), **with respect to** recall, precision and f-measure, **from the viewpoint of** developers and researchers, **in the context of** detecting design and requirement self-admitted technical debts in open source projects.

4.2 Planning

Context Selection: We selected the dataset discussed in Section 3.1 for the classification of SATDs. Just as in previews works, use only design and requirement SATDs to train the Word2vec model, and train and test the LSTM model. The model was validated using a leave one-out cross-project validation approach to two dataset groups. We trained the models using 9 of the 10 projects and tested on the remaining one. This procedure is repeated 10 times so that each project can be tested with the trained model.

Hypothesis Formulation: To reach the proposed goal, we define the following research question: Is the LSTM neural network with Word2vec better than previous works in terms of recall, precision, F-measure?

The following hypothesis was defined for each proposed metric, h_0: the algorithms have the same metric mean (Eq. 7) and h_1: the algorithms have distinct metric means (Eq. 8). Note that h_0 is the hypothesis that we want to refute.

$$\mu_1(metric) = \mu_2(metric) \tag{7}$$

$$\mu_1(metric) \neq \mu_2(metric) \tag{8}$$

Selection of Participants: We divided the dataset into two groups, the first (60,907 code comments) having comments with design SATDs and comments without any SATDs, and the second (58,961 code comments) with comments with requirement SATDs and comments without SATDs.

Experiment Project: The experiment project refers to the following stages: Preparation of the development environment, it means downloading and installation of all the items described in instrumentation. Subsequently, we implement and trained the model with the dataset. Finally, we run the experiments and perform statistical tests for the assessment of the defined hypotheses.

Independent Variables: The LSTM neural network, Word2vec model and the dataset.

Dependent Variables: Predictions made by the model, represented by: precision (Eq. 9), recall (Eq. 10) and f-measure (Eq. 11). True positives (TP) are cases in which the classifier correctly identifies a SATD comment and true negative (TN) corresponds to the correct classification of a comment without SATD. If the model classifies a SATD comment as without SATD, it is a case of false negative (FN), and the case of false positive (FP) is when the model classifies a comment without SATD as a SATD comment.

$$precision = \frac{TP}{TP + FP} \tag{9}$$

$$recall = \frac{TP}{TP + FN} \tag{10}$$

$$fmeasure = \frac{2 \times precision \times recall}{precision + recall} \tag{11}$$

Instrumentation: The instrumentation process started with the environment configuration for the achievement of the controlled experiment; data collection planning; and the development and execution of the assessed algorithms. The used materials/resources were: Keras (Fraçois et al, 2015), Scikit-learn (Pedregosa et al, 2011), Gensim (Rehurek and Sojka, 2010), and a computer with Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz, 16 GB RAM - 64 bits. The preparation of the test environment was done by downloading and installing all the mentioned libraries.

4.3 Execution

After all preparation, the experiment was performed. First, the dataset was loaded and a cleanup process was performed. Some special characters and numbers were eliminated so as not to confuse the training process and to improve the quality of the features. Then the data was submitted to the training model. At the end, leave-one-out cross-project validation was carried out on the 10 projects.

4.4 Data Validation

We used two statistical tests to validate our results: Student's t-test and Shapiro-Wilk test. Student's t-test is used to determine if the difference between two means is statistically significant. For this, it is necessary that the distribution of samples is normal. Normality is tested using Shapiro-Wilk.

5 RESULTS

After performing the training and testing of the LSTM model, the results of the classification were obtained through the leave-one-out cross-project validation process. Table 2 and 3 present the results achieved for each project and metric in the design classification SATDs and requirement SATDs respectively. We use the abbreviations for precision (Pr), recall (Rc), and f-measure (F1) because of the little space available in the table. The best result for each metric is highlighted in bold.

The results show that without pre-processing with Word2vec, LSTM has higher recall than Auto-sklearn (AS) and Maximum Entropy (ME), but loses in precision and f-measure. When pre-processing with Word2vec is applied, we have a significant improvement in precision, and consequently in the f-measure. The model without word embedding showed a 56% improvement in recall compared to Auto-sklearn and Maximum Entropy in the design SATD classification. With the application of Word2vec, the improvement in recall was approximately 36% in both cases. Despite having a lower recall, when we apply a pre-processing with word embedding, the precision of the LSTM network increases by approximately 135%. For design SATD, the Auto-sklearn classifier from

Table 2: Comparison of the metrics obtained in the design SATD classifications.

	LSTM + word2vec			LSTM			Auto-sklearn			Maximum entropy		
	Pr	Rc	F1	Pr	Rc	F1	Pr	Rc	F1	Pr	Rc	F1
Ant	0.621	0.608	**0.614**	0.228	**0.821**	0.357	**0.676**	0.301	0.360	0.554	0.484	0.517
ArgoUML	**0.940**	0.798	**0.863**	0.443	**0.963**	0.607	0.784	0.703	0.741	0.788	0.843	0.814
Columba	0.658	0.813	0.728	0.148	0.952	0.256	0.765	**0.940**	**0.842**	**0.792**	0.484	0.601
EMF	0.346	0.613	0.442	0.069	**0.910**	0.129	**0.802**	0.501	**0.604**	0.574	0.397	0.470
Hibernate	0.735	**0.912**	**0.814**	0.467	0.873	0.609	**0.833**	0.450	0.583	0.877	0.645	0.744
JEdit	0.316	**0.837**	0.459	0.214	0.744	0.333	**0.943**	0.701	**0.810**	0.779	0.378	0.509
JFreeChart	0.418	0.733	**0.532**	0.277	**0.885**	0.422	**0.872**	0.250	0.390	0.646	0.397	0.492
JMeter	0.734	**0.859**	**0.791**	0.233	0.854	0.367	0.706	0.420	0.530	**0.808**	0.668	0.731
JRuby	0.845	0.838	**0.841**	0.362	**0.932**	0.522	**0.856**	0.750	0.801	0.798	0.770	0.784
Squirrel	0.545	0.708	**0.616**	0.192	**0.894**	0.317	**0.903**	0.630	0.740	0.544	0.536	0.540
Average	0.616	0.772	**0.670**	0.263	**0.882**	0.391	**0.814**	0.564	0.640	0.716	0.560	0.620

Table 3: Comparison of the metrics obtained in the requirement SATD classifications.

	LSTM + word2vec			LSTM			Auto-sklearn			Maximum entropy		
	Pr	Rc	F1	Pr	Rc	F1	Pr	Rc	F1	Pr	Rc	F1
Ant	0.230	0.500	**0.315**	0.013	**0.692**	0.026	**0.650**	0.136	0.226	0.154	0.154	0.154
ArgoUML	**0.839**	0.683	0.753	0.388	**0.854**	0.533	0.779	0.762	**0.771**	0.663	0.540	0.595
Columba	**0.860**	0.755	0.804	0.107	**1.000**	0.194	0.781	0.935	**0.851**	0.755	0.860	0.804
EMF	0.375	**0.750**	0.500	0.015	0.562	0.030	**0.826**	0.682	**0.747**	0.800	0.250	0.381
Hibernate	0.765	0.765	0.765	0.165	0.921	0.281	0.809	0.435	0.566	0.610	0.391	0.476
JEdit	0.571	0.666	0.615	0.014	**0.785**	0.028	**0.937**	0.715	**0.811**	0.125	0.071	0.091
JFreeChart	0.800	0.266	0.400	0.064	**0.800**	0.118	**0.846**	0.280	**0.421**	0.220	0.600	0.321
JMeter	0.619	0.565	**0.590**	0.029	**0.952**	0.057	**0.693**	0.418	0.522	0.153	0.524	0.237
JRuby	0.572	**0.900**	0.700	0.296	0.763	0.427	**0.859**	0.749	**0.800**	0.686	0.318	0.435
Squirrel	0.780	0.513	0.619	0.060	**0.760**	0.112	**0.848**	0.535	**0.656**	0.657	0.460	0.541
Average	0.414	0.636	0.606	0.115	**0.809**	0.180	**0.803**	0.565	**0.637**	0.482	0.416	0.403

Wattanakriengkrai et al. (2018) obtained the best results in precision.

The good results achieved in the SATD design classification were not maintained in the SATD requirement classification. The SATD requirement dataset has a lower number of positive cases, which makes it difficult to train the LSTM network. In this case, the f-measure results were statistically compatible between the LSTM and Auto-sklearn models. The Auto-sklearn classifier was superior in precision, but produced a lower recall than LSTM models with and without word embedding.

Although the LSTM model has a higher average recall and f-measure, it was necessary to follow a statistical validation to verify if the improvement was significant. The next step was to perform the Shapiro-Wilk test with the set of metrics. The Shapiro-Wilk test showed that the distribution of the metrics is normal. In this way, we applied the Student's t-test for paired samples to verify if the difference was statistically significant.

Table 4, 5, and 6 presents the p-values calculated from the average precision, recall and f-measure respectively obtained with the classification models for design and requirement SATDs detection. As can be seen, the improvements in recall and f-measure in the design SATD classification and improvement in recall in requirement SATD have p-values lower than the significance level of 0.05. This indicates that only for these cases the improvement was statistically significant. The reason for this may be related to the lower amount of requirement SATDs for training the LSTM network, which shows that the LSTM network is dependent on a larger dataset for better results.

Table 4: Results from t-test for average precision.

LSTM + word2vec vs Classifiers	Design		Requirement	
	p-value	Result	p-value	Result
Maximum entropy	0.11	Retain H_0	0.11	Retain H_0
Auto-sklearn	0.03	Refute H_0	0.03	Refute H_0
Only LSTM	0.00	Refute H_0	0.00	Refute H_0

Table 5: Results from t-test for average recall.

LSTM + word2vec vs Classifiers	Design		Requirement	
	p-value	Result	p-value	Result
Maximum entropy	0.00	Refute H_0	0.05	Refute H_0
Auto-sklearn	0.01	Refute H_0	0.23	Retain H_0
Only LSTM	0.01	Refute H_0	0.03	Refute H_0

Table 6: Results from t-test for average f-measure.

LSTM + word2vec vs Classifiers	Design		Requirement	
	p-value	Result	p-value	Result
Maximum entropy	0.01	Refute H_0	0.00	Refute H_0
Auto-sklearn	0.66	Retain H_0	0.47	Retain H_0
Only LSTM	0.00	Refute H_0	0.00	Refute H_0

5.1 Threats to Validity

There are some aspects of an experiment that define the validity of the results achieved during its execution. It is ideal that all threats to the validity of the experiment are known and that measures are taken to have them reduced or eliminated. The following are threats found during the planning and execution of the experiment:

1. Construct validity
    a. The implementation of an LSTM neural network algorithm must meet the theoretical requirements and any changes may compromise its results. To ensure that a correct implementation of the LSTM neural network was evaluated, we used the Keras (Fraçois et al, 2015) library that has thousands of citations in study publications;
    b. A manually annotated dataset may contain errors caused by human failure, such as incorrect labelling and labelling bias. This may compromise classifier performance. In this case, we compared the LSTM model with other classifiers that used the same dataset and followed the same process of validation of the experiment.

6 CONCLUSION AND FUTURE WORKS

In this work, we evaluated a neural network LSTM model with Word2vec in the classification of design and requirement self-admitted technical debts through a controlled experiment. The results were compared with two other natural language processing approaches: auto-sklearn and maximum entropy classifiers.

At the end of the experiment, it was possible to verify that the LSTM model improved the recall and f-measure in design SATDs classification and recall in requirement SATD. The average f-measure was statistically the same as the model with better precision, in this case the Auto-sklearn classifiers.

Recall may be more important than precision depending on the problem being discussed (Hand and Christen, 2018). Someone may accept a slightly higher rate of false positives to get more true positives, if the trade-off is considered interesting.

The LSTM model without Word2vec has a better recall rate but lower precision. Combining Word2vec and LSTM brings a great advantage to the overall performance.

In future works, others neural networks and deep learning architectures can be evaluated in this context. There are results that show that the combination of convolutional neural networks and LSTM achieve good results in the task of text (Zhou et al, 2015). In addition, more in-depth research should be done to find ways to reduce the amount of false negatives produced by the LSTM model in this dataset. Overall, we found that the LSTM model and word2vec can outperform other models.

REFERENCES

Cunningham, W. (1992). The WyCash portfolio management system. ACM SIGPLAN OOPS Messenger, 4(2), 29-30.

Seaman, C., & Guo, Y. (2011). Measuring and monitoring technical debt. In Advances in Computers (Vol. 82, pp. 25-46). Elsevier.

Potdar, A., & Shihab, E. (2014, September). An exploratory study on self-admitted technical debt. In 2014 IEEE International Conference on Software Maintenance and Evolution (pp. 91-100). IEEE.

Guo, Y., & Seaman, C. (2011, May). A portfolio approach to technical debt management. In Proceedings of the 2nd Workshop on Managing Technical Debt (pp. 31-34).

Codabux, Z., & Williams, B. (2013, May). Managing technical debt: An industrial case study. In 2013 4th International Workshop on Managing Technical Debt (MTD) (pp. 8-15). IEEE.

Nord, R. L., Ozkaya, I., Kruchten, P., & Gonzalez-Rojas, M. (2012, August). In search of a metric for managing architectural technical debt. In 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (pp. 91-100). IEEE.

Marinescu, R. (2012). Assessing technical debt by identifying design flaws in software systems. IBM Journal of Research and Development, 56(5), 9-1.

Alves, N. S., Ribeiro, L. F., Caires, V., Mendes, T. S., & Spínola, R. O. (2014, September). Towards an ontology of terms on technical debt. In 2014 Sixth International Workshop on Managing Technical Debt (pp. 1-7). IEEE.

Maldonado, E. D. S., & Shihab, E. (2015, October). Detecting and quantifying different types of self-admitted technical debt. In 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD) (pp. 9-15). IEEE.

da Silva Maldonado, E., Shihab, E., & Tsantalis, N. (2017). Using natural language processing to automatically detect self-admitted technical debt. IEEE Transactions on Software Engineering, 43(11), 1044-1062.

Wattanakriengkrai, S., Maipradit, R., Hata, H., Choetkiertikul, M., Sunetnanta, T., & Matsumoto, K. (2018, December). Identifying design and requirement self-admitted technical debt using n-gram idf. In 2018 9th International Workshop on Empirical Software Engineering in Practice (IWESEP) (pp. 7-12). IEEE.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. nature, 521(7553), 436-444.

Zhou, C., Sun, C., Liu, Z., & Lau, F. (2015). A C-LSTM neural network for text classification. arXiv preprint arXiv:1511.08630.

Zhou, P., Qi, Z., Zheng, S., Xu, J., Bao, H., & Xu, B. (2016). Text classification improved by integrating bidirectional LSTM with two-dimensional max pooling. arXiv preprint arXiv:1611.06639.

Hand, D., & Christen, P. (2018). A note on using the F-measure for evaluating record linkage algorithms. Statistics and Computing, 28(3), 539-547.

Huang, Q., Shihab, E., Xia, X., Lo, D., & Li, S. (2018). Identifying self-admitted technical debt in open source projects using text mining. Empirical Software Engineering, 23(1), 418-451.

Liu, Z., Huang, Q., Xia, X., Shihab, E., Lo, D., & Li, S. (2018, May). Satd detector: A text-mining-based self-admitted technical debt detection tool. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (pp. 9-12).

Young, T., Hazarika, D., Poria, S., & Cambria, E. (2018). Recent trends in deep learning based natural language processing. ieee Computational intelligenCe magazine, 13(3), 55-75.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). Experimentation in software engineering. Springer Science & Business Media.

Basili, V. R., & Weiss, D. M. (1984). A methodology for collecting valid software engineering data. IEEE Transactions on software engineering, (6), 728-738.

François, C. et al. (2015). Keras, https://keras.io

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. Journal of machine learning research, 12(Oct), 2825-2830.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.

Mikolov, T., Chen, K., Corrado, G. S., & Dean, J. A. (2015). U.S. Patent No. 9,037,464. Washington, DC: U.S. Patent and Trademark Office.

Rehurek, R., & Sojka, P. (2010). Software framework for topic modelling with large corpora. In In Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks.