Parallel Implementation of a Bug Report Assignment Recommender Using Deep Learning

Adrian-Cătălin Florea[1(✉)], John Anvik[2], and Răzvan Andonie[1,3]

[1] Electronics and Computers Department, Transilvania University of Braşov, Braşov, Romania
acflorea@unitbv.ro

[2] Department of Mathematics and Computer Science, University of Lethbridge, Lethbridge, AB, Canada
john.anvik@uleth.ca

[3] Computer Science Department, Central Washington University, Ellensburg, WA, USA
andonie@cwu.edu

Abstract. For large software projects which receive many reports daily, assigning the most appropriate developer to fix a bug from a large pool of potential developers is both technically difficult and time-consuming. We introduce a parallel, highly scalable recommender system for bug report assignment. From a machine learning perspective, the core of such a system consists of a multi-class classification process using characteristics of a bug, like textual information and other categorical attributes, as features and the most appropriate developer as the predicted class. We use alternatively two Deep Learning classifiers: Convolutional and Recurrent Neural Networks. The implementation is realized on an Apache Spark engine, running on IBM Power8 servers. The experiments use real-world data from the Netbeans, Eclipse and Mozilla projects.

1 Introduction

Bug report triage is the process by which bug reports submitted to a software project's "bug tracker", a form of issue tracking system, are analyzed to determine if the report will result in development activity. As large software development projects can receive hundreds of bug reports per day [4,5], bug report triage is a significant software maintenance problem, as it can take substantial time and resources [8]. Bug report assignment is an important bug report triage decision, as errors can result in delays to the project [4,5,7].

Deep Learning techniques are a modern tool that provides highly promising results in a vast area of applications [10]. They have the drawback of long training times and complex architectures which often do not scale well. The high complexity of such models has so far lead to their low adoption rate in the area of recommender systems.

We present a highly scalable parallel Deep Learning-based implementation of a bug report assignment recommender system. To the best of our knowledge,

A. Lintas et al. (Eds.): ICANN 2017, Part II, LNCS 10614, pp. 64–71, 2017.
https://doi.org/10.1007/978-3-319-68612-7_8

no previous attempts have been made in using Deep Learning techniques for bug report assignment recommendations. Our main focus is not the execution time itself, but the scalability of the system on a cluster. This is measured by the speedup (the ratio of the sequential execution time to the parallel execution time) and parallel efficiency (speedup divided by the number of processors/cores). In a cluster architecture, these are important metrics. Our contributions are the following:

- *The first Deep Learning-based approach for automated bug report assignment in large software projects*, with prediction performance on par with the state of the art.
- *A highly scalable, parallel implementation* that alleviates the problem of a long training time and model size.

The paper proceeds as follows: Sect. 2 gives a brief overview of the existing work in the context of Deep Learning for recommender systems and previous attempts of using machine learning techniques for bug report assignment recommendation. We then describe our recommender system in Sect. 3. In Sect. 4, we discuss the results obtained using three real-world datasets from the Netbeans, Eclipse and Mozilla projects. The paper is concluded in Sect. 5.

2 Related Work

We restrict our focus to Deep Learning architectures used to train recommender systems. The following are some of the results which are related to the recent high interest in Deep Learning.

Salakhutdinov *et al.* [17] proposed one of the first approaches, where they integrate an instance of a two-layer Restricted Boltzmann Machines (RBM) into a Collaborative Filtering type recommender to be able to handle large datasets. They tested their model on the Netflix dataset, consisting of over 100 million user/movies rating. Although the approach outperformed the existing models at that time, it used as input exclusively the ratings and it was found to be not deep enough. Wang *et al.* [20] proposed a hierarchical Bayesian model called Collaborative Deep Learning (CDL) and used movie review text along with the ratings to alleviate the cold-start problem. Reviews were converted into a numerical representation using the Bag Of Words (BOW) technique, and this numerical data were fed into a Deep Auto Encoder to obtain a lower dimensional feature space [19]. The model gave good results, but it suffered from losing the semantic information embedded in the text, as well as from ignoring the word order.

There have been other attempts to integrate different types of Deep Neural Networks into recommender systems. As a potential solution to deal with the cold start problem, Wang *et al.* used a Convolutional Neural Network (CNN) to develop a hybrid recommender that considers real-world users information and high-level representation of audio data [21]. Kyo-Joong *et al.* [16] developed a personalized news recommender system based on a three-layer perceptron in which they integrated several numerical features extracted from textual data.

Tomar *et al.* [18] used a Word2Vec [6] model to provide hashtag recommendations for English tweets[1]. The obtained word representation was then fed into a deep feedforward neural network.

Given the large amount of data, as well as the large number of parameters required to build deep learning models, leveraging the resources of a cluster to optimize the training is gaining a lot of attention. With Apache Spark offering a dense and intuitive API for building parallel applications, combining Spark and Deep Learning is an emerging research direction. Stoica *et al.* [13] developed the SparkNet framework that uses Apache Spark to train multiple instances of Caffe models, distributed over a cluster using data parallelism[2]. Even though it is not a recommender system per se, SparkNet can provide the backbone of such a system. Similar attempts of using Apache Spark to parallelize deep learning were also reported by Abu Alsheikh [2]. Their work focused on using Deep Learning for mobile data analytics.

In the area of bug report assignment recommendation, standard machine learning techniques have been employed, including Naïve Bayes [14], SVM [9], and C4.5 [3]. Some of these techniques were also used in combination with dimensionality reduction, such as LSI [1], X^2 [5,9] and LDA [9,15]. The first parallel bug report assignment recommender systems was implemented by Florea *et al.* [9] and used a distributed version of a SVM on a Spark cluster.

3 Deep Learning Bug Report Assignment Recommender

This section presents our recommender system. As shown in Fig. 1, the input to the recommender system consists of MySQL database dumps. The data is filtered and pre-processed as described in Sect. 3.2. We export the pre-processed data as input for building a Paragraph2Vec [12] model which we later feed into either a CNN or a Long Short Term Memory (LSTM) network [10].

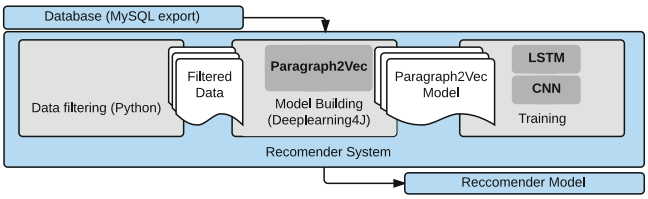

Fig. 1. Recommender system macro architecture.

We describe in the following sections: the datasets used for evaluation, the cleansing and pre-processing phases, the training algorithm, and the implementation.

[1] https://twitter.com/.
[2] https://en.wikipedia.org/wiki/Data_parallelism.

3.1 Datasets

To evaluate the performance of our recommender system, we use real data from the following three software projects: **Eclipse** and **Netbeans** - both datasets available from the 2011 Working Conference on Mining Software Repositories Data Challange webpage,[3] and **Mozilla** - made available to us by the Mozilla Foundation. All of the datasets were in the form of anonymized MySQL dumps of the respective Bugzilla database. From these data dumps, we extract data from the fields of the following tables: `bugs` (`bug_id`, `creation_ts`, `bug_status`, `product_id`, `resolution`, `component_id`, `bug_severity`), `bugs_activity` (`who`, `bug_when`), `longdescs` (`bug_id`, `thetext`) and `duplicates` (`dupe`). The `bugs` table stores essential details of a bug including its id, severity, creation date, and current bug status. The `bugs_activity` table stores the activity logs, including all of the changes in the bug status. The `duplicates` table contains information about duplicate bug reports. The bug report's textual information is kept in the `longdescs` table.

3.2 Data Preparation

To prepare the data, we apply similar data cleansing steps as described by Anvik and Murphy [4] and Florea *et al.* [9]. Among all the CLOSED bugs (`bugs.bug_status` = `'CLOSED'`), we consider exclusively the reports marked as FIXED (`bugs.resolution` = `'FIXED'`). Based on project heuristics, as detailed by Anvik and Murphy [4], we consider the developer who fixed a report as the one marking it as FIXED in `bugs_activity`. To be able to produce effective predictions, we restrict our input both in terms of training data and in terms of target classes (developers). We consider a developer to be active in the project if, and only if, she fixed an average of three or more bug reports per month over the last three months. Doing so eliminates developers with only occasional contributions to the project. We also remove those that have an exceptionally high frequency of marking reports FIXED (more than $mean + 2 * stddev$), on the assumptions that these represent project managers and not developers. For each bug report, we use a normalized version of its textual representation obtained by converting the text to lower case, removing any non-alphanumerical characters, and concatenating the text from the report's title, description, and comments. In addition to textual data, we use the `component_id`, `product_id` and `bug_severity` fields as one-hot-encoded categorical variables.

We use the Distributed Bag of Words version of Paragraph Vector (PV-DBOW) [12] to obtain numerical values for features from our textual data. For each dataset, we train a model with 100 features for 20 epochs, with no hard-coded stopwords, using a window size of 5. The input for the PV-DBOW model is the text retrieved from all of the reports in the database. This is done as each term can contain valuable semantic information, regardless of the report status, resolution or age. For training, we restrict the data to bugs marked as

[3] http://2011.msrconf.org/msr-challenge.html.

FIXED during the most recent 240 days, considering that the most recent data is also the most relevant data.

3.3 Recommender Training

We are interested in predicting recent data based on older one. Therefore, we order the data based on most recent change date and split it into 80% training, 10% validation and 10% test. We tune the recommender using the validation data and report the results obtained on the test data. We train the recommender on an Apache Spark[4] cluster using either a CNN or a LSTM network.

Convolutional Neural Network. To induce a level of spatiality between words from a certain paragraph, for CNN, we split the textual representation of bugs into `n` equal sequences and compute their PV-DBOW representation individually. Our network of choice consists of two pairs of convolution and average pooling layers, followed by two dense layers, as shown in Fig. 2. The "C" layers $(C(k_i, 1)(1, 1)f_i)$ represent convolution layers with a $(k_i, 1)$ size kernel, a stride of $(1, 1)$ and f_i filters. "P" layers are pooling layers, while "D" layers are dense layers. For our recommender system, we set the value of `n` to five and (k_i) to two for both of the "C" layers. The first dense layer has 500 neurons and uses ReLU $(f(x) = max(0, x))$ activation. For the second dense layer, we use the softmax $(\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}})$ activation.

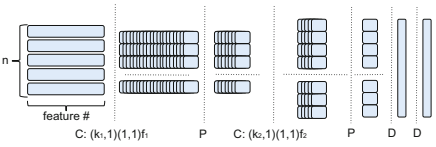

Fig. 2. The CNN architecture

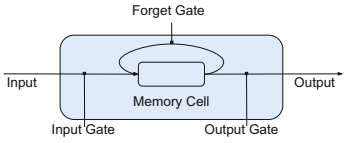

Fig. 3. An LSTM cell

Long Short Term Memory. For LSTM, no further data preprocessing is required. The data is fed to a two-layer Recurrent Neural Network. The first layer is a "Graves" [11] RNN unidirectional layer with 250 output neurons, a dropout of 0.5 and the softsign $(f(x) = \frac{x}{1+|x|})$ activation. Each cell is a typical memory cell as shown in Fig. 3. The output layer uses the softmax activation.

3.4 Implementation Details

The recommender system is developed in Scala 2.11.8, with the exception of the data cleansing phase, which is implemented in Python 2.7.11. We use Deeplearning4j[5] to build and train the neural networks. We train both the CNN and LSTM

[4] http://spark.apache.org/.
[5] https://deeplearning4j.org/about.

networks on an Apache Spark engine. Each Spark cluster consists of one master node and 4, 8 or 12 workers. Data is distributed among the workers and the training starts with the initial set of parameters. Once every n iterations, the new parameter values are sent back to the master which averages and re-distributes them to the worker nodes. Training continues with the new parameter values and the process repeats for a predefined number of steps or until certain convergence criteria are met.

We train and test our recommender on two IBM Power8 8001-22C (Briggs) servers with Ubuntu 16.04 LTS installed, using the IBM. version of Spark 1.6.3. Each server has 2 processors with 20 cores each and 512GB of RAM. The code is publicly available on GitHub[6].

4 Results

Table 1 shows the precision, recall and F1-measure values obtained by the LSTM and CNN recommenders compared to the distributed SVM-based results reported in Florea *et al.* [9]. The CNN recommender version shows comparable results across all projects in terms of classification performance when compared to the parallel SVM implementation. Although LSTM has, in most of the cases, a

Table 1. Precision, recall and F1-measure after 100 epochs with batch sizes of 10 and 25 samples compared to the parallel SVM implementation. Best values are in bold.

Dataset	Netbeans					Eclipse					Mozilla				
Architecture	LSTM		CNN		SVM	LSTM		**CNN**		SVM	LSTM		CNN		SVM
Batch size	10	25	10	25	-	10	25	10	25	-	10	25	10	25	-
Precision	0.86	0.85	**0.90**	0.88	0.89	0.73	0.67	**0.80**	**0.80**	0.78	0.59	0.62	**0.78**	0.77	0.77
Recall	0.83	0.83	0.87	0.86	**0.88**	0.70	0.68	0.75	0.75	**0.77**	0.70	0.71	**0.75**	**0.75**	**0.75**
F1-measure	0.84	0.84	**0.88**	0.87	**0.88**	0.71	0.67	**0.77**	**0.77**	**0.77**	0.64	0.66	**0.76**	0.75	0.73

Table 2. Network average training time per epoch (in seconds), speedup and efficiency with increasing number of workers as compared to a 4 cores execution baseline.

Dataset	Netbeans				Eclipse				Mozilla			
Architecture	LSTM		CNN		LSTM		CNN		LSTM		CNN	
Batch size	10	25	10	25	10	25	10	25	10	25	10	25
Average training time	408	212	442	175	1192	492	1411	525	3347	1028	3636	1579
Speedup 8 cores	1.83	2.93	2.85	1.84	2.02	2.14	2.34	2.16	2.84	1.66	2.01	1.82
Efficiency 8 cores	0.92	1.47	1.42	0.92	1.01	1.07	1.17	1.08	1.42	0.83	1.00	0.91
Speedup 12 cores	2.53	4.06	4.31	4.12	2.52	2.69	3.96	3.67	5.23	3.12	2.94	3.06
Efficiency 12 cores	1.27	2.03	2.15	2.06	1.26	1.35	1.98	1.84	2.62	1.56	1.47	1.53

[6] https://github.com/acflorea/deep-columbugus,mariana-triage.

significantly lower training time, as shown in Table 2, it yields worse performance results for all projects. Of the four sample sizes (10, 25, 100 and 250) examined, 100 and 250 samples led to notably worse classification performance for LSTM when compared to sample sizes of 10 or 25. However, for CNN, increasing the sample size resulted in only a small performance degradation.

Both network architectures have good scalability. The speedup and efficiency values for training the recommender are depicted in Table 2.

5 Conclusions

A highly scalable parallel Deep Learning-based implementation of a bug report assignment recommender system was introduced, with both the CNN and LSTM approaches explored. Using a parallel implementation, we trained the recommender relatively fast, considering that deep network training is generally slow. Although the CNN recommender achieves results on par with the parallel SVM implementation, the SVM approach remains superior in terms of training speed.

This work opens further research directions both in terms of optimizing the training speed (e.g., use GPUs instead of CPUs) and prediction performance (e.g., identify more efficient CNN architectures).

Acknowledgments. The authors are grateful to the Mozilla Foundation for providing a dump of their Bugzilla database and to IBM Client Center, Poughkeepsie, NY, USA for allowing us to use their infrastructure.

References

1. Ahsan, S.N., Ferzund, J., Wotawa, F.: Automatic software bug triage system (bts) based on latent semantic indexing and support vector machine. In: Fourth International Conference on Software Engineering Advances, ICSEA 2009, pp. 216–221, September 2009
2. Alsheikh, M.A., Niyato, D., Lin, S., Tan, H., Han, Z.: Mobile big data analytics using deep learning and apache spark. CoRR abs/1602.07031 (2016). http://arxiv.org/abs/1602.07031
3. Anvik, J., Hiew, L., Murphy, G.C.: Who should fix this bug? In: Proceedings of the 28th International Conference on Software Engineering, ICSE 2006, NY, USA, pp. 361–370 (2006). http://doi.acm.org/10.1145/1134285.1134336
4. Anvik, J., Murphy, G.C.: Reducing the effort of bug report triage: recommenders for development-oriented decisions. ACM Trans. Softw. Eng. Methodol. **20**(3), 10:1–10:35. http://doi.acm.org/10.1145/2000791.2000794
5. Banitaan, S., Alenezi, M.: Tram: an approach for assigning bug reports using their metadata. In: 2013 Third International Conference on Communications and Information Technology, pp. 215–219, June 2013
6. Bengio, Y., Ducharme, R., Vincent, P., Janvin, C.: A neural probabilistic language model. J. Mach. Learn. Res. **3**, 1137–1155. http://dl.acm.org/citation.cfm?id=944919.944966

7. Bhattacharya, P., Neamtiu, I., Shelton, C.R.: Automated, highly-accurate, bug assignment using machine learning and tossing graphs. J. Syst. Softw. **85**(10), 2275–2292 (2012)
8. Cavalcanti, Y.A.C., da Mota Silveira Neto, P.A., do Carmo Machado, I., de Almeida, E.S., de Lemos Meira, S.R.: Towards understanding software change request assignment: a survey with practitioners. In: Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering, pp. 195–206 (2013)
9. Florea, A.-C., Anvik, J., Andonie, R.: Spark-based cluster implementation of a bug report assignment recommender system. In: Rutkowski, L., Korytkowski, M., Scherer, R., Tadeusiewicz, R., Zadeh, L.A., Zurada, J.M. (eds.) ICAISC 2017. LNCS, vol. 10246, pp. 31–42. Springer, Cham (2017). doi:10.1007/978-3-319-59060-8_4
10. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge (2016). http://www.deeplearningbook.org
11. Graves, A.: Supervised Sequence Labelling with Recurrent Neural Networks. Studies in Computational intelligence, New York (2012). http://opac.inria.fr/record=b1133792
12. Le, Q.V., Mikolov, T.: Distributed representations of sentences and documents. CoRR abs/1405.4053 (2014). http://arxiv.org/abs/1405.4053
13. Moritz, P., Nishihara, R., Stoica, I., Jordan, M.I.: SparkNet: training deep networks in spark. ArXiv e-prints, November 2015
14. Nasim, S., Razzaq, S., Ferzund, J.: Automated change request triage using alpha frequency matrix. In: Frontiers of Information Technology (FIT), pp. 298–302 (2011)
15. Nguyen, T.T., Nguyen, A.T., Nguyen, T.N.: Topic-based, time-aware bug assignment. SIGSOFT Softw. Eng. Notes **39**(1), 1–4. http://doi.acm.org/10.1145/2557833.2560585
16. Oh, K.J., Lee, W.J., Lim, C.G., Choi, H.J.: Personalized news recommendation using classified keywords to capture user preference. In: 16th International Conference on Advanced Communication Technology, pp. 1137–1155 (2014)
17. Salakhutdinov, R., Mnih, A., Hinton, G.: Restricted boltzmann machines for collaborative filtering. In: Proceedings of the 24th International Conference on Machine Learning, ICML 2007, NY, USA, pp. 791–798 (2007). http://doi.acm.org/10.1145/1273496.1273596
18. Tomar, A., Godin, F., Vandersmissen, B., Neve, W.D., de Walle, R.V.: Towards twitter hashtag recommendation using distributed word representations and a deep feed forward neural network. In: 2014 International Conference on Advances in Computing, Communications and Informatics, Delhi, India, September 24–27, 2014 (2014). http://dx.doi.org/10.1109/ICACCI.2014.6968557
19. Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., Manzagol, P.A.: Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. J. Mach. Learn. Res. **11**, 3371–3408. http://dl.acm.org/citation.cfm?id=1756006.1953039
20. Wang, H., Wang, N., Yeung, D.: Collaborative deep learning for recommender systems. CoRR abs/1409.2944 (2014). http://arxiv.org/abs/1409.2944
21. Wang, X., Wang, Y.: Improving content-based and hybrid music recommendation using deep learning. In: Proceedings of the 22nd ACM International Conference on Multimedia, MM 2014, NY, USA, pp. 627–636 (2014). http://doi.acm.org/10.1145/2647868.2654940