UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Atakan Arıkan

Issue Resolution Time Prediction Using Deep Learning Techniques

Master's Thesis (30 ECTS)

Supervisor(s): Dietmar Pfahl

Tartu, 2019

Issue Resolution Time Prediction Using Deep Learning Techniques

Abstract:

Issue resolution time prediction has a large importance in software projects since planning of these projects are typically hard. Especially in the agile practices, such as sprint planning, to be able to predict correctly how long it would take to resolve an issue, holds the power to plan correctly.

This thesis focuses on the state of the art approaches to this problem and study their performances. On top of that we discuss how can one structure and implement a deep learning algorithm to solve issue resolution time prediction problem. Afterwards, we compare and discuss the results of the applied deep learning technique with the current state of the art.

The data used for this study contains around 700,000 issues. This data is gathered collectively from the previous studies in this field. By using the already existing data, we plan to validate the existing results and build on top of the current baseline.

Keywords:

Issue resolution, defect resolution time, issue time prediction, machine learning, deep learning, neural networks, resolution time, software issue, prediction, GitHub, data analysis,

CERCS: P170 Computer science, numerical analysis, systems, control

Probleemi lahendamise ajakulu prognoosimine süvaõppe abil

Lühikokkuvõte:

Probleemi lahendamise ajakulu prognoosimine on tarkvaraprojektide korral suure tähtsusega, kuna selliste projektide planeerimine on raske. Probleemi lahendamisele kuluva aja täpne hindamine on eriti vajalik agiilses tarkvaraarenduses, nt sprindi planeerimises, sest see võimaldab planeerida täpselt. Käesolev magistritöö keskendub antud probleemi kaasaegsetele lähenemisviisidele ja nende efektiivsuse uurimisele. Töös arutletakse selle üle, kuidas on võimalik struktureerida ja rakendada süvaõppe algoritme probleemi lahendamisele kuluva aja prognoosimiseks. Süvaõppe meetodite abil saadud tulemusi võrreldakse teiste kaasaegsete tulemustega. Andmed, mida antud lõputöös kasutatakse, sisaldavad umbes 700 000 probleemi. Andmed on kogutud kollektiivselt samas valdkonnas varem läbiviidud uurimustest. Kasutades olemasolevaid andmeid, on töös plaanis valideerida olemasolevaid tulemusi ja neid täiendada.

Võtmesõnad:

Probleemi lahendamine, veaparanduse aeg, probleemi lahendamise ajakulu prognoosimine, masinõpe, süvaõpe, tehisnärvivõrgud, lahendusaeg, tarkvara probleem, prognoosimine, GitHub, andmeanalüüs

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

List of Abbreviations

DNN	Deep Neural Network
IRT	Issue Resolution Time
VCS	Version Control System
FFNN	Feed-Forward Neural Network
LSTM	Long-Short Term Memory
RNN	Recurrent Neural Network
CNN	Convolutional Neural Network
RF	Random Forest
DT	Decision Trees
CFS	Correlation Feature Selection
TP	True Positive
TN	True Negative
FP	False Positive
FN	False Negative
ReLU	Rectified Liner Unit

1. Introduction

Companies use planning to structure their course of action in order to achieve their goals in a long or short run. During planning, one must value the importance and/or value of an achievement and the effort it would take to actually accomplish it. Every business decision is made by comparing the effort and value of taking or not taking that specific action. With our very fast modern day, companies make business decision on a daily, even hourly basis. Due to its nature of dealing with future events, planning has a very close relation to prediction. During planning, one always predicts the time it would take to achieve a goal, and the result of that accomplishment. If the predictions made during the planning phase was not structured well, then the result of the plan is highly unlikely to turn as predicted.

Nowadays, the size of the software projects has grown so much that it is nearly impossible to build a software without a plan. Since it would also be really hard to plan building and maintaining the whole software directly, developers and product managers tend to break big chunks of features, or software descriptions into smaller tasks and subtasks. The reasoning behind this is to minimise the risk factor induced by the size of the ticket and being able to correctly estimate how long a ticket would take to implement.

Especially after the introduction of the agile manifesto, planning in software engineering started to focus on smaller tasks rather than large chunks of issues. Sprint planning became a vital part of software development. And since these sprints are generally of fixed length, people needed to know how many issues that they could fit in a sprint. However, just looking at issues and predicting how long it would take to solve them is a really hard task, since there could be many factors affecting the outcome, such as environmental factors or recently discovered bugs that have more importance. Being able to predict how long it would take to resolve an issue, based on some selected features would give both the developers, product managers and company owners huge advantages in planning their future decisions. They could measure the importance of an issue based on the predicted time it would take to resolve it, meaning even if the resolution of the issue would bring a big business value to the company, possible time spent on the issue might cancel the aforementioned business value.

A typical way how companies handle this problem is to use the concept of story points in their development cycle. Which is almost always non unanimous and biased based on each developers own experience. The only purpose of the story points is to give programmers and product managers a vague idea about how large the issue is, rather than actually predicting how long it would take to resolve the issue.

Now that we are entering the age of data, it is only logical to make use of generated data and make predictions about our future. In this study, a neural network is going to be trained to predict the resolution time of any given issue, based on the dataset we gathered

from an existing study[1]. However, the results of this study would be applicable to other applications and datasets, given nature of the neural networks.

1.1 Problem Statement

The aim of this study is to apply a deep learning technique on a given set of issues and compare the results with already existing studies.

Since the data used in this study differs from the ones used in previous studies, a need for replicating the already existing work has surfaced. This replication is crucial for us to make any comparison between two approaches.

Therefore, the main questions that drives the purpose of this thesis are:

1. How can one select and implement a deep learning algorithm to predict issue resolution time?
2. How does a deep neural network perform against other conventional classification algorithms (i.e. Decision Trees and Random Forest)?

1.2 Study Outline

The structure for the remaining of this thesis is given below:

- Section 2 provides a general overview of the state of the art in terms of issue resolution time prediction, while explicitly stating the search strategy. In addition, we quickly introduce the selected baseline study for our comparison purposes.
- Section 3 describes our dataset, baseline selection and deep learning implementation details
- Section 4 shows sand explain the results from the experiments on the datasets from Section 3.
- Section 5 discusses the results and show a comparison between the two approaches.
- Section 6 introduces a prediction tool which can be used in enterprise systems to predict an issues lifetime, upon its creation.
- Section 7 concludes the thesis.

2. Related Work

Given the importance of software in our daily and business needs, the IRT prediction problem has been gaining a lot of attention from researchers. For the purpose of researching the existing studies, We accessed Google Scholar, IEEE and ACM digital libraries, since they are considered to be trustworthy resources.

As a starting point, we used publications from Pfahl et al. from year 2015 and 2016, which were related to issues in version control systems. In addition to this, we conducted queries on these digital libraries using the keywords "issue resolution, issue resolution time prediction, defect resolution, issue lifetime, issue resolution using deep learning".

Using the methods mentioned above, we were able to collect 6 papers as starting points. One of those papers focused on improving expert prediction of the IRT. These experts could be a developer or a software architect or a any other stakeholder with enough experience and knowledge, therefore we left it our of the scope of this study. One masters thesis from a student in University of Tartu did a study on IRT prediction on the issues collected from an Estonian company, whose data was private. This resulted in us disregarding the paper, since it would be impossible to validate the results of the paper, without access to the data. After reading the abstract of the remaining four, we realised one of them was focused on issue dynamics, studying how often the issues are generated, average lifetime and so on, rather than the IRT problem. This left us with three papers. Reading these papers and traversing through their references resulted in two more studies on this topic.

Adding specific algorithm names to our previous search query, such as "decision trees, neural networks, random forest and deep learning" lead us to another paper that we haven't encountered before, which studied prediction of issue resolution time using a decision tree model using issues from different projects. This paper is also added to the list of selected literature, since its closely related to this topic.

Overall, the selected papers for the literature survey are the following:

- Panjer L. D. (2007)
- Bhattacharya, P., & Neamtiu, I. (2011)
- Assar, S., Borg, M., & Pfahl, D. (2016)
- Kikas, R., Dumas, M., & Pfahl, D. (2016)
- Rees-Jones, M., Martin, M., & Menzies, T. (2017).

One of the first approaches to the IRT problem is from Panjer L. D.[2] which was released in 2007. He studied the data from the Eclipses Bugzilla project. Using around 120,000 issues, he compares the results of different approaches, namely *0-R, 1-R, C.45 Decision Tree, Naive Bayes* and lastly *Logistic Regression.* He uses 15 features in this study, such as *severity, dependencies, version, assigned developer* and so on.

Looking at his results, the algorithms perform around 30% accuracy on the average, calculated by using cross-validation. Logistic Regression performs the best with 0.349 accuracy, whereas 0-R performs the worst with 0.291 accuracy. He also states that the most influential features were the comment activity, severity of the bug and the project that the bug was assigned to.

In 2011 a paper titled *Bug-fix Time Prediction Models: Can We Do Better?*[3] from *Pamela Bhattacharya* and *Iulian Neamtiu*, the authors are also only focused on bug type of issues, disregarding the other types such as features, stories, epics and so on. They provide a statistical approach to test whether the bug report features, which were being used by the state of the art approaches of the time, were related to the bug-fix time or not. Therefore the scope of the aforementioned paper is a bit limited compared to ours. However, the results they provided in their paper affected the led the next researchers in a correct way. They collect data from four different projects, namely Firefox, Seamonkey, Eclipse and Thunderbird, resulting in over 500,000 bug reports. Then test with the existing approaches to find a correlation between the bug report fields and the dependent variable, bug-fix time.

Their regression analysis show that the approaches which use features such as severity of the bug, number of attachments on the bug, number of developers working on the bug have zero to none correlation with the resulting bug-fix time. They also studied whether reputation of the bug opener, which is basically the number of issues this particular developer closed, divided by the number of issues they are assigned, has a very low correlation for the result, contrary to the findings of Guo et al.[4] and Hooimeijer et al. [5].

In their 2016 paper, Said Assar, Markus Borg, and Dietmar Pfahl explore a text-based clustering algorithm to predict the issue resolution time. Their work is an extension of the 2013 paper from Uzma Raja[6], where they replicate their work on a different data and setting. Then they continue to provide their algorithm and compare the results of the two.

Kikas et al. use a different approach in their paper from 2016. They use both static and dynamic features in prediction of the issue lifetime. Static features are the ones which are available upon the issues creation, such as body of the issue, number of issues created by the author at that time, severity of the issue, project the issue is related to and so on. Dynamic features are added after the issue is created, namely number of comments, added text to the issue body and so on.
The data used in this study is collected from GHTorrent[7], a third party software that tracks and collect public data from the popular VCS website, GitHub. They collected around almost a million issues, which are created between years 2012 and 2014, from more than

4,000 projects. 30 % of the issues collected are labeled as *sticky,* meaning that they are not closed in the observation period they selected. For those issues, They consider the end of their lifetime as 1st of January, 2015, which is the end date for their issue collection approach.

They then train multiple Random Forest[8] classifiers to do the prediction. The way the prediction made in their paper is the following, they choose 7 observation points (0, 1, 7, 14, 30, 90, and 180 days), and 7 prediction horizons (1, 7, 14, 30, 90, 180 and 365 days). They follow it by training the models, which would predict whether an issue, on a given observation point, would be closed in the given prediction horizon. And since it is meaningless to create a model which would have a prediction horizon smaller than its observation point, they end up with 28 models. They display their results in terms of AUC and F1 scores. Their best performing models are when they predict the closing time of a recently opened issue, meaning *observation point 0.* The results are especially better for cases with long lifetimes, such as 90, 180 and 365 days. The model that predicts whether a *newborn* issue will be closed more or less than 365 days has the best AUC and F1 scores, with 0,707 and 0,898 respectively. Their overall results show a correlation with the number of issues from the given timeclass used to train model. Meaning the model performance is closely related to the number of *positive* issues fed to the model upon training.

Menzies et al. used simple decision tree models in their 2017 paper[9]. Their paper takes Kikas et al.'s work as a baseline, and compare their results with Kikas et al.'s approach. They argue that simple models, with less amount of features would perform better in this topic. On that topic, they choose to use only static features from Kikas et al.'s paper, since they make their prediction on issue lifetime, rather than predicting based on an *observation point* and *prediction horizon*. Therefore, they prune the 21 features listed in the 2016 paper to 8, as 7 input features and 1 output feature. Then they created N-binary models for 5 buckets (1, 7, 14, 30 and 90 days). Each of these models predicts whether given issue would be closed in *N* days or not.

Menzies et al. choose to use CFS[10] in order to select the best features from the selected eight before training the model. They state that using CFS yields in better predictors. After this step, they train their models using the C4.5 algorithm. Their results are evaluated with two different approaches:

- Using cross-validation on locally-trained trees, where a 10-fold cross-validation is applied on each project on its own data.
- Round robin approach on cross-project trees, where for each selected project, training with the remaining data from all other projects and predicting with the current project.

They state results for both approaches, and in general, cross-project training seems to outperform the local training by a margin of 10-15%.

3. Methodology

3.1. Structure

This section explains the overall structure of this study.

Firstly, we justify the reasoning behind why we chose our dataset. Then explain how we derived multiple datasets and how they were made suitable for the experiment we had in mind. Then we explain the distribution of the time class in these multiple datasets.

Next section articulates the overall case experiment setup of this study. We go over the baseline algorithm, how and why we selected it and how the results from that study were derived. Following that section, we introduce our deep learning implementation to tackle the IRT problem. This part shows what are the options are there for the deep learning algorithm, state in which fields which algorithm is best and overall justify the selection of our deep learning algorithm.

After stating the experiment setup and implementation details, next section displays the results from all the experiments in this study. We discuss the results of each experiment in their own sections.

Discussion of the results is made in the next section. There we articulate why the results turned out the way they are, what could be the implications and how does the structure of the data affect the results.

Figure 1 shows a flow chart that represents the overall methodology of this study.

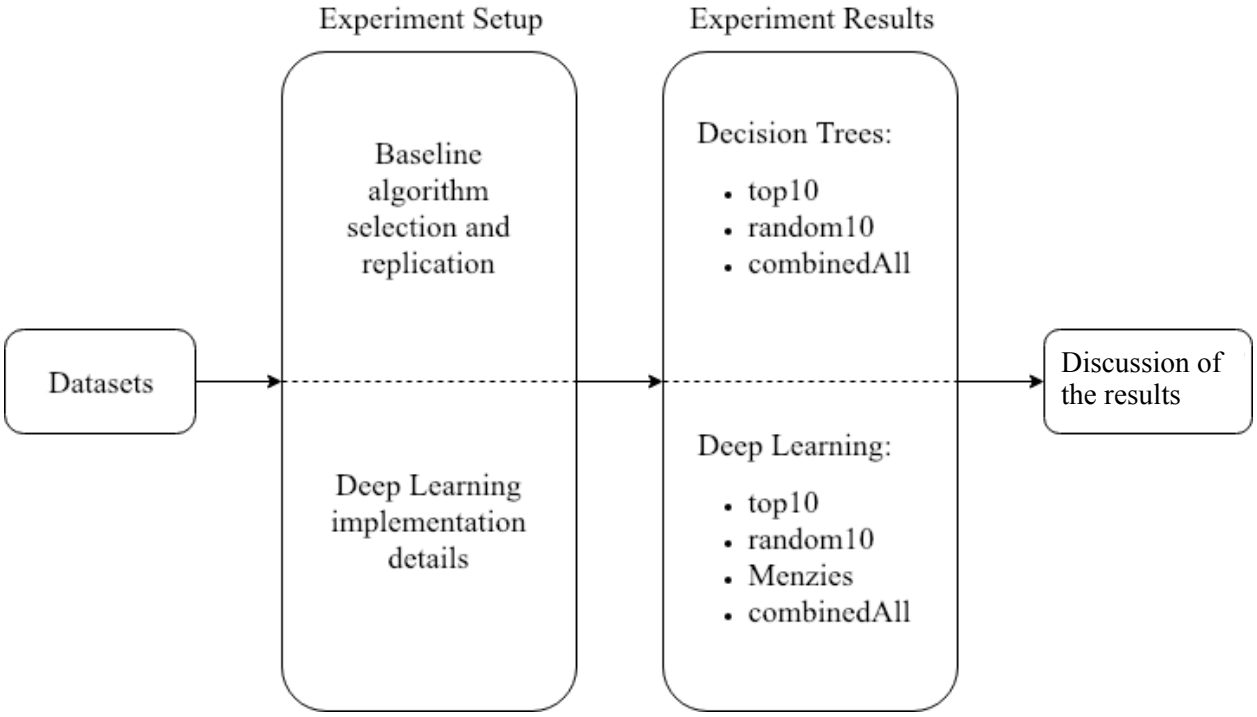

Figure 1: Outline of the methodology of this study

3.2 Datasets

The dataset for this study was chosen to be the same data from Kikas et al.'s study. They collected around 900.000 issues from over 6000 repositories. We selected this dataset for this study because it was available, it contained a lot of issues for training a deep learning model and it contained the same features used in the study that we wanted to compare our results with.

After gathering the data, we needed to transform it to make it applicable for Menzies et al.'s algorithm. Similar to the Menzies et al.'s approach, we also did not consider how long the given issue was open for, therefore we decided to disregard the dynamic features from our dataset. Since in their paper, Menzies et al. choose to ignore *sticky* issues, we had to filter them out from the dataset as well. Also, the issues in our dataset were not separated with regards to the projects they belonged to, but rather they were together in one place. After also separating them into different files for the projects they belonged, we were ready to train the algorithm with this data.

A small problem we encountered before running any experiments and doing any comparison was that our dataset contained issues from more than 4,000 different repositories, whereas Menzies et al.'s data contained issues from only 10 repositories. And since train-

ing more than 20,000 models and studying the results would not be feasible, we selected two sets of 10 projects from our dataset.

We repeated our experiments for the cases where we selected 10 projects with most issues, called Top 10 repositories. Then we selected 10 random projects, named Random 10 Repositories. All repositories under each group are listed below:

10 top repositories

- appium
- Font-Awesome
- framework
- hd
- Ionic
- ppsspp
- RedisDesktopManager
- SCII-External-Maphack
- steam-for-linux
- wet-boew

10 randomly selected repositories

- Cataclysm-DDA
- custard
- docs
- fuzzy-octo-tribble
- openDCIM
- partake
- PushSharp
- pythondotorg
- rbb
- slycat

On top of these data subsets, we also wanted to test how Menzies et al.'s approach performs against a large dataset. At the end, we compared the results between four combined datasets:

- **combinedAll**: Contains all the data in our dataset
- **combinedMenzies**: Contains combined data from Menzies et al.'s paper
- **combinedRandom10**: Contains combined data from randomly selected dataset
- **combinedTop10**: Contains combined data from 10 repositories with the most issues

Among these datasets, as expected, *combinedAll* has the largest number of issues with 657622 issues in total. Since the second largest dataset, that is *combinedMenzies,* has 41174 issues, we chose not to include the former dataset in the comparison plot. *combinedTop10* dataset has 26149 issues in total, whereas the smallest set, *combinedRandom10* has 2478 issues in total. *Table 1* displays the information regarding the number of issues and their classes for each dataset.

	Total number of issues			
	combinedAll	**combinedMenzies**	**combinedTop10**	**combinedRandom10**
< 1	275168	7279	12269	1088
1 - 7	113973	4894	4420	423
7 - 14	53263	1747	1829	181
14 - 30	58265	2212	2304	201
30 - 90	74002	5219	2816	229
> 90	82951	19823	2511	356
TOTAL	**657622**	**41174**	**26149**	**2478**

Table 1: Total number of issues and their distribution in each datasets.

Figure 2 shows the distribution of the *timeOpen* feature over the combined datasets. *combinedMenzies* has a very skewed distribution towards the class 90, as in issues that are closed after 90 days. The graph for *combinedTop10* repositories seems a bit evenly distributed compared the combinedMenzies, however, classes 1 and 90 still dominate the others. In case of the *combinedRandom10*, the distribution seems a bit flatter, compared to the others, however the total number of issues is quite low. *combinedAll* dataset seems to have the same structure as the other datasets which were derived from it, namely *combinedTop10* and *combinedRandom10*.

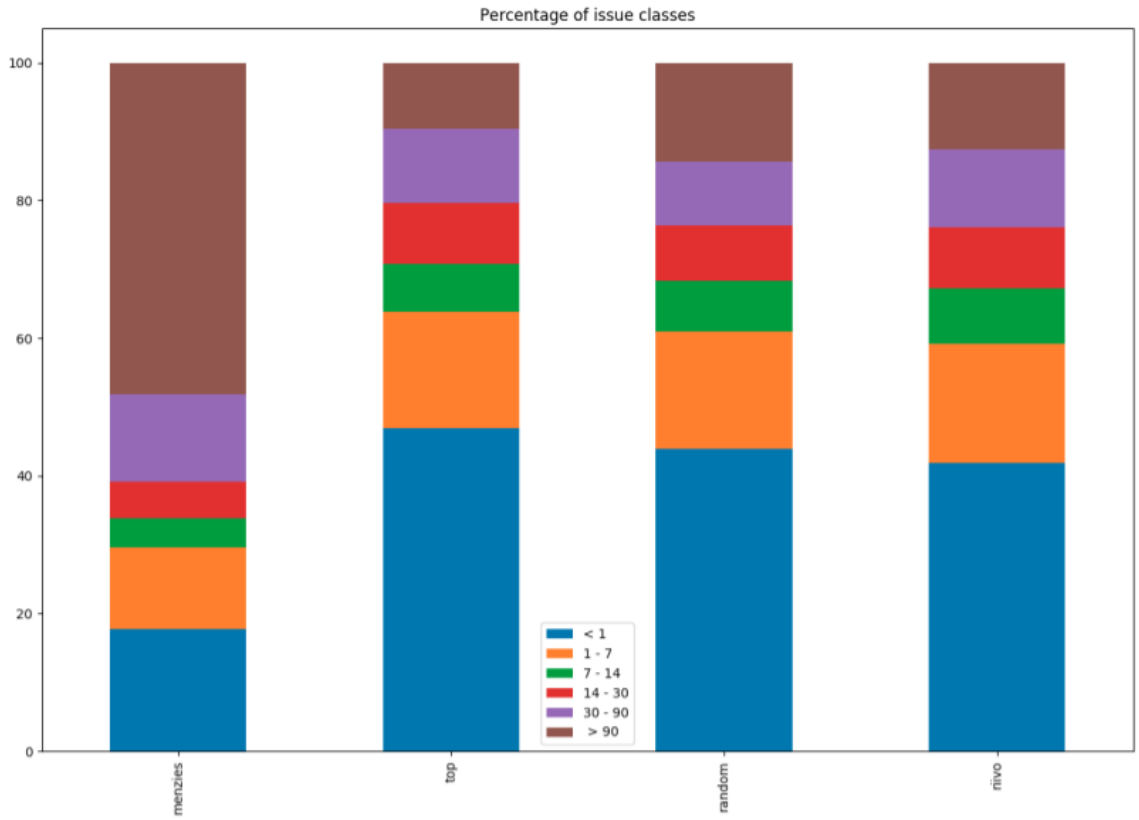

Figure 2: Percentage of time classes for each of the combined datasets.

3.3 Baseline

By the time of the preparation of this study, the two main and most successful approaches to IRT problem were from Kikas et al. and Menzies et al., namely random forests and decision trees. Menzies et al. compared the results from their decision trees implementation to Kikas et al.'s random forest implementation. They claimed and showed that simple classifiers such as decision trees performed better than complex classifiers like random forests.

However, since Menzies et al. used a different dataset in their studies, we first wanted to validate that decision trees in fact outperformed the random forest implementation. As mentioned before, Kikas et al.'s random forest implementation considers how long an issue has been alive, and makes a prediction based on that. Therefore in their paper, Kikas et al. listed results for all possible combination. However, for the sake of being able to make a comparison between this study and Menzies et al.'s study, we chose to only consider the case where the issues have been open for 0 days, meaning that they were just opened. After collecting Kikas et al.'s results, we executed Menzies et al.'s decision trees algorithm on Kikas et al.'s data to see whether decision trees did outperform the random forest algorithm.

Table 2 shows the performances of the two algorithms. In four out of five cases Decision Tree (DT) algorithm performs better than the Random Forest (RF) algorithm. Therefore, we selected Menzies et al.'s decision tree algorithm as our baseline and ran our experiments on this algorithm.

	F1 Values for Decision Trees	**F1 Values for Random Forest**
1	0,302	0,437
7	0,758	0,604
14	0,816	0,659
30	0,876	0,715
90	0,947	0,781

Table 2: F1 scores of both algorithms ran on the combinedAll dataset

The paper articulates how they executed two different types of experiments, using the same algorithm. First, they used a 10-fold cross-validation on each projects data, training and predicting with issues from the same project. On top of this, they also trained models for each project, using data from the remaining projects for training, and tested the model with the issues from the current project, which they called a round robin experiment.

After collecting the publicly available data and code, we needed to validate that we could reproduce the same results from the aforementioned paper.

Since our plan was to compare the results of both the cross validation and round robin experiments, we ran both of their experiments on the same data from the paper. Since the code and the data were the same with the paper, we expected to get very similar results from both of the experiments. For the cross validation experiment, we were able to reproduce the similar results without encountering any problems. The results from the original paper and our replication can be seen from Table 3. Since the context of the experiment is different than the one in the paper, we expected to see some minor differences. Looking at the results, we accepted that the results were similar, and the further results from this algorithm would be reliable.

For the case of the round robin experiment however, the results were not similar. We debugged the given code to get rid of any trivial bugs, however we fail to find any. We contacted the developers of the algorithm, but sadly we were not able to get a response.

Since the results from the round robin experiment were not reliable, we decided to leave the round robin experiment from the scope of this study. Therefore, in our experiments, we only used the cross validation method to display and compare the performances of the algorithms.

		Menzies et al.'s Results			**Our Replication Results**		
		precision	**recall**	**pf**	**precision**	**recall**	**pf**
camel	1	65	40	4	65	42	4
	7	74	70	7	74	69	7
	14	73	78	10	73	80	10
	30	82	80	9	83	80	9
	90	89	70	5	89	71	5
cloudstack	1	66	60	24	66	52	22
	7	76	93	77	77	93	76
	14	81	96	83	81	97	85
	30	85	100	100	85	100	100
	90	94	100	100	94	100	100

cocoon	1	0	0	0	0	0	0
	7	0	0	0	0	0	0
	14	0	0	0	0	0	1
	30	53	96	14	53	96	12
	90	67	95	11	67	95	11
deeplearning	1	78	77	41	76	77	41
	7	88	100	100	88	100	100
	14	86	100	100	86	100	100
	30	91	100	100	91	100	100
	90	96	100	100	96	100	100
hadoop	1	0	0	0	0	0	0
	7	0	0	0	0	0	0
	14	0	0	0	0	0	0
	30	0	0	0	0	0	0
	90	32	2	0	34	4	1
hive	1	0	0	0	0	0	0
	7	0	0	0	0	0	0
	14	52	35	1	52	35	1
	30	0	0	0	0	0	0
	90	62	53	9	62	63	8
kafka	1	63	48	17	62	49	18
	7	78	83	43	77	83	45
	14	81	90	56	81	89	55
	30	83	97	76	83	98	76
	90	91	98	71	91	98	71
node	1	56	31	16	56	31	16
	7	69	95	89	69	95	89
	14	76	100	100	76	100	100
	30	84	100	100	84	100	100
	90	93	100	100	93	100	100
ofbiz	1	0	0	0	0	0	0
	7	54	43	27	56	45	27
	14	56	70	57	56	69	56
	30	62	87	77	62	87	77
	90	67	100	100	68	98	100
qpid	1	0	0	0	0	0	0
	7	0	0	0	0	0	0
	14	0	0	0	0	0	0
	30	0	0	0	0	0	0
	90	53	19	5	56	24	6

Table 3: Cross validation results from the original paper and our replication

3.4 Deep Learning

This section articulates the details of our deep learning implementation. Firstly, we justify the reasons behind our algorithm selection. Then the details of our model are given.

3.4.1 Algorithm selection

We used a feed forward neural network for our deep neural network implementation. Before settling on the this decision, we tried out multiple types of other neural networks, such as Convolutional Neural Networks, Recurrent Neural Networks and Long Short Term Memory Neural Networks. Next section explains why the decision of FFNN was made.

Firstly, we implemented a CNN model to fit our data with. CNNs are quite powerful in areas where the input layer contains many features. Image processing is a prime example for this. Since an image is represented as a matrix of pixels, the input layer of the model results in *NxM* nodes, where *N* and *M* show the resolution of the image. And most of the time, the nodes that are close to each other are semantically connected. In case of an image, adjacent pixels do not differ from each other that much. And when they do, it means the shape that is displayed on the image ends in that pixel. CNNs are especially great at recognising these *patterns* using a method called *convolution*, which makes use of the large number of input layers. CNNs take the input layers and apply filters on the nodes. These filters are small matrixes that iterate over the input layer and generate the next layers by doing matrix multiplication.

Our data was not applicable to these type of networks since it only had 7 nodes in the input layer. Designing a filter that would iterate over a 7 x 1 matrix was not the best idea. Even if it was, the nodes in our input layer were not related to each other by any means. Therefore a CNN would not be able to recognise a pattern between the nodes. Due to aforementioned reasons. implementing a CNN was not applicable to our dataset.

Apart from CNNs, we tried to create an LSTM Neural Network model for our prediction purposes. LSTMs are a kind of RNNs which looks at the historical data when doing back propagation. LSTMs are extremely powerful for *time series* datasets. These types of networks have an inner *historical state*, that affects the outcome of the prediction of the model. In turn, this prediction updates the historical state of the LSTM network. Due to this property, they have a representation of a *memory,* hence the name *long short term memory* neural networks.

Another area where LSTMs excel is in Natural Language Processing. Machine translation or prediction of the next word in a given sentence are among the areas where LSTMs or RNNs in general are widely used. However, our dataset was again not really applicable to

this approach. The dataset was not really a time series data, and there was no repeating pattern. For example, in a text data, it would be really common to see the token ‘am’ after the token ‘I’, and while predicting the next word, the model would make use of this fact. Or, if we were to do stock price prediction, looking back at the last *N* entries would allow us to make a better prediction, since tomorrows stock price would be heavily dependant on todays stock price.

One might think that our dataset could be represented as time series, however, the time the issues are created or closed are not dependant on each other. An argument could be that if there is already some number of issues were open during the time this new particular issue was alive, it might affect the lifespan of this new issue. However, there is very little information on how the actual development power was spent on these set of issues. We simply can’t know if the whole team stopped working on a feature request since there was an urgent bug, or only one person worked on a bug, even though the issue has only one concerned stakeholder. On top of this, an issue generally does not go through its development lifecycle upon creation. That means, when an issue is created, it does not automatically affect the other issues in the repository immediately.

3.4.2 Model Implementation

In order to implement the selected deep learning approach, we chose to use a Python library called Keras[11]. It is a simple yet very powerful python library that provides human readable APIs, and has a very well written documentation. Since its quite popular, the community feedback is also available. Rather than implementing some boilerplate code, or complex structures, Keras has already built-in functions that are very useful to developers and researchers. On top of this, it also utilises other popular deep learning languages and libraries such as TensorFlow[12] and Theano[13].

Our neural network has seven input layers, one for each feature that we have. For the hidden layers, we found out that a dense layer of 16 nodes, with an Rectified Linear Unit (ReLU) activation, repeated twice gave the best results. The output layer was using Sigmoid activation and had only one node. When predicting, we assumed any value more than 0.5 belonged to the class 1, and anything lower belong to the class 0.

Our optimiser was chosen to be Adam[14], with a learning rate of 0.001, which seemed to perform really well with numeric inputs in prediction tasks. And for our loss function, we used binary cross-entropy, since we were basically doing an N-binary classification problem, for different target classes.

4. Results

In the tables shown in this section, whenever a cell is filled with red background means that the data is *bad*, meaning false alarms(*pf*) are over 33% and recall and precision values are below 33%. This convention follows the one introduced in Menzies et al.'s paper. The first column shows the results for the dataset and the second columns displays how many issues belonged to which class. Since we are using N-binary classifiers, we can use positive or negative classes. For every experiment, **positive** means that the issue was closed before the target class, whereas **negative** means that the issue was closed after than the target class. For example, if the target class is 7, an issue would belong to class **positive** if it was closed in less than 7 days. With the same logic, if it was closed in more than 7 days, it would be in the **negative** class.

The definitions of the *precision, recall* and *pf* are given next. However, before explaining those, one should understand the underlying concepts, such as:

- **True positives (TP):** Issue was closed in less than *N* days, classifier said it will be closed in less than *N* days.
- **False positives (FP):** Issue was closed after *N* days, classifier said it will be closed in less than *N* days.
- **True negatives (TN):** Issue was closed after *N* days, classifier said it will be closed after *N* days.
- **False negatives (FN):** Issue was closed in less than *N* days, classifier said it will be closed in after *N* days.

Where *N* is the target class. Knowing those, the definitions of precision, recall and pf are the following:

- $precision = TP / (TP + FP)$
- $recall = TP / (TP + FN)$
- $pf = FP / (TN + FP)$

4.1 Decision Trees Implementation Results

In this section, we display the results from each of our experiments using the decision trees algorithm, on the datasets described in Section 3.2. We execute the DT algorithm on each of the Top 10 repositories, then on each of the Random 10 repositories. Lastly, we run the algorithm on each of the combined datasets.

4.1.1 Experiment 1: Cross Validation on *Top10* Dataset

The results of this experiment can be seen in Table 4. We see a general increase in precision and recall as time class goes larger for most of the repositories, except for some edge

cases in repositories like ionic and ppspp. This result is expected as the ratio of positive to negative issue count increases with each time class.

		Cross Validation			Issue Counts	
		precision	recall	pf	Positive	Negative
appium	1	57	14	8	923	1180
	7	59	99	96	1236	867
	14	65	99	95	1379	724
	30	76	99	94	1585	518
	90	91	100	100	1899	204
Font-Awesome	1	83	95	45	1743	783
	7	88	94	40	1860	666
	14	89	92	39	1938	588
	30	91	93	39	2020	506
	90	93	96	53	2215	311
framework	1	65	79	57	1692	1209
	7	74	96	87	2102	799
	14	79	98	87	2246	655
	30	85	98	84	2418	483
	90	96	97	69	2734	167
hd	1	68	69	26	1563	2001
	7	84	93	38	2436	1128
	14	93	94	24	2736	828
	30	96	97	25	3090	474
	90	98	100	100	3470	94
ionic	1	73	1	0	786	1137
	7	60	100	100	1166	757
	14	68	100	100	1330	593
	30	81	100	100	1556	367
	90	95	100	100	1826	97
ppsspp	1	66	32	14	1197	1327
	7	59	85	79	1475	1049
	14	62	99	98	1588	936
	30	67	100	100	1732	792
	90	80	100	100	2048	476
RedisDesktopManager	1	75	62	6	604	2100
	7	87	88	13	1313	1391
	14	90	96	22	1818	886
	30	94	98	43	2368	336
	90	97	100	90	2696	8

SCII-External-Maphack	1	81	86	25	1901	1394
	7	91	99	33	2565	730
	14	94	100	27	2691	604
	30	94	88	59	2943	352
	90	99	100	100	3269	26
steam-for-linux	1	60	63	34	817	888
	7	74	88	53	1103	602
	14	79	87	53	1215	490
	30	80	96	87	1359	346
	90	87	100	100	1501	204
wet-boew	1	56	9	5	1043	1291
	7	63	87	82	1433	901
	14	68	100	100	1577	757
	30	74	100	100	1751	583
	90	85	100	100	1980	354

Table 4: Cross validation results for decision trees on Top10 repositories.

Figure 3 shows the averages of precision, recall and pf for each of the repositories. The distribution and the results for this dataset shows that the decision trees algorithm performs really well when the data is balanced.

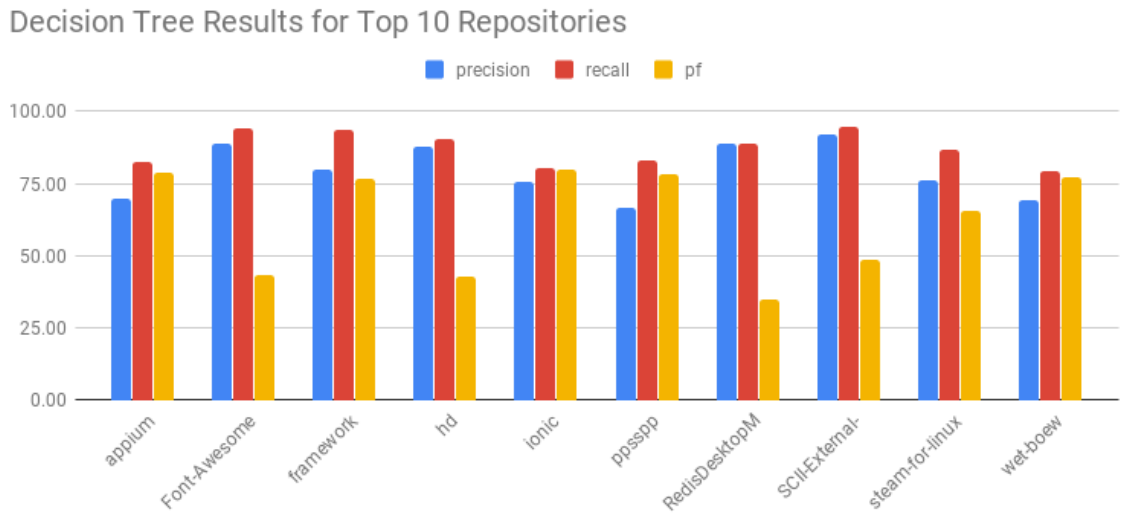

Figure 3: Averages for pf, precision and recall for all time classes on Top 10 repositories

The issue time class distribution per repository can be found in Figure 4. *RedisDesktop-Manager* repository has good *precision* and *recall* while having a small *pf* value. And it has a fairly even distribution in terms of time classes. On top of this, having almost no issues that are closed in more than 90 days seems to help as well. Another well performer is

SCII-External-Maphack, which also has zero to none issues that were closed after 90 days. While this increases the precision and percentage, it also leads to a fairly biased classifier, therefore increasing the value of *pf*.

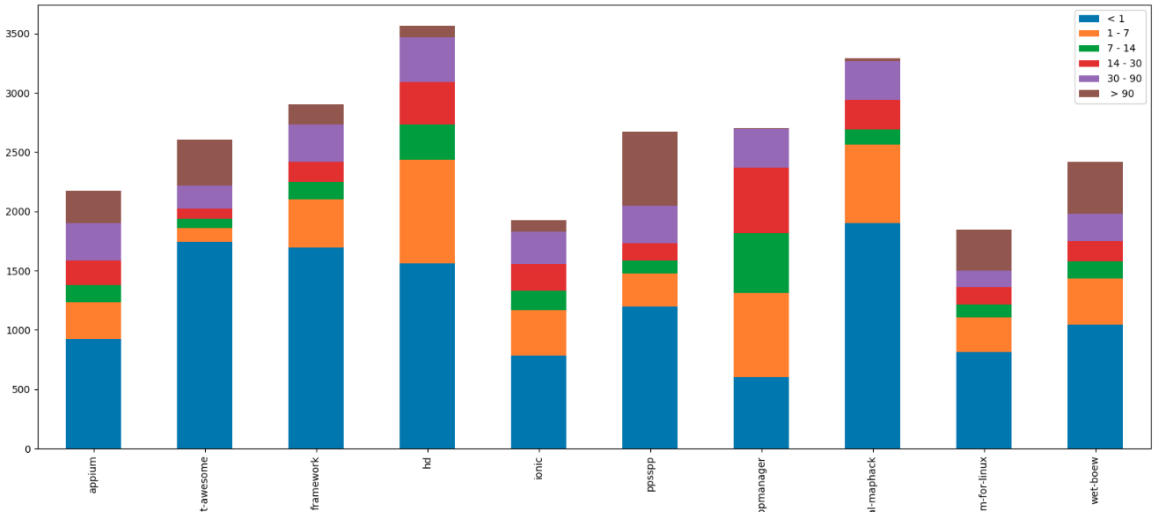

Figure 4: Time class distribution of the issues of each Top 10 repository

4.1.2 Experiment 2: Cross Validation on *Random10* Dataset

The results of this experiment can be seen in Table 5. Similarly to the previous experiment, we see correlation between time class and precision, with a couple of exceptions. Especially in partake repository, which has very little issues overall, and no issues that were closed between the period of 14 to 30 days, which is reflected in the precision and recall values for those time classes.

		Cross Validation			Issue Counts	
		precision	recall	pf	Positive	Negative
Cataclysm-DDA	1	76	65	14	78	109
	7	85	78	31	131	56
	14	78	97	93	147	40
	30	87	97	97	165	22
	90	99	100	80	186	1
custard	1	0	0	0	89	293
	7	65	17	6	152	230
	14	58	75	53	195	187
	30	76	56	31	238	144
	90	78	92	70	271	111
docs	1	48	60	44	147	195
	7	57	81	65	186	156
	14	71	78	54	218	124
	30	79	92	83	268	74
	90	88	100	100	302	40

fuzzy-octo-tribble	1	75	95	65	191	80
	7	93	97	88	247	24
	14	94	98	92	251	20
	30	97	98	96	258	13
	90	100	100	100	271	0
openDCIM	1	54	87	85	201	160
	7	67	100	100	249	112
	14	77	94	96	280	81
	30	85	100	100	306	55
	90	94	100	100	335	26
partake	1	100	71	0	5	24
	7	47	46	26	10	19
	14	0	0	25	11	18
	30	0	0	25	11	18
	90	61	100	100	17	12
PushSharp	1	0	0	0	81	121
	7	60	97	98	125	77
	14	63	92	98	138	64
	30	74	98	100	153	49
	90	91	100	100	183	19
pythondotorg	1	53	100	100	124	112
	7	67	100	100	155	81
	14	69	100	100	161	75
	30	74	97	98	179	57
	90	84	98	100	196	40
rbb	1	61	66	34	72	96
	7	85	92	44	115	53
	14	90	89	42	130	38
	30	93	91	37	136	32
	90	93	97	74	149	19
slycat	1	47	66	51	100	128
	7	59	99	97	141	87
	14	70	99	98	161	67
	30	77	99	100	179	49
	90	95	100	100	212	16

Table 5: Cross validation results for random forest on Random10 repositories.

Just like in the previous dataset, the repositories that have somewhat even distribution resulted in smaller pf values. For instance, if we compare *Cataclysm-DDA* and *rbb* repositories, even though they have similar amount of issues, and similar amounts of *precision* and *recall*, the *pf* values differ because the classifier for *rbb* has seen at least some issues that

were closed in more than 90 days. Figure 5 shows the averages for pf, precision and recall for each time class in Random 10 dataset.

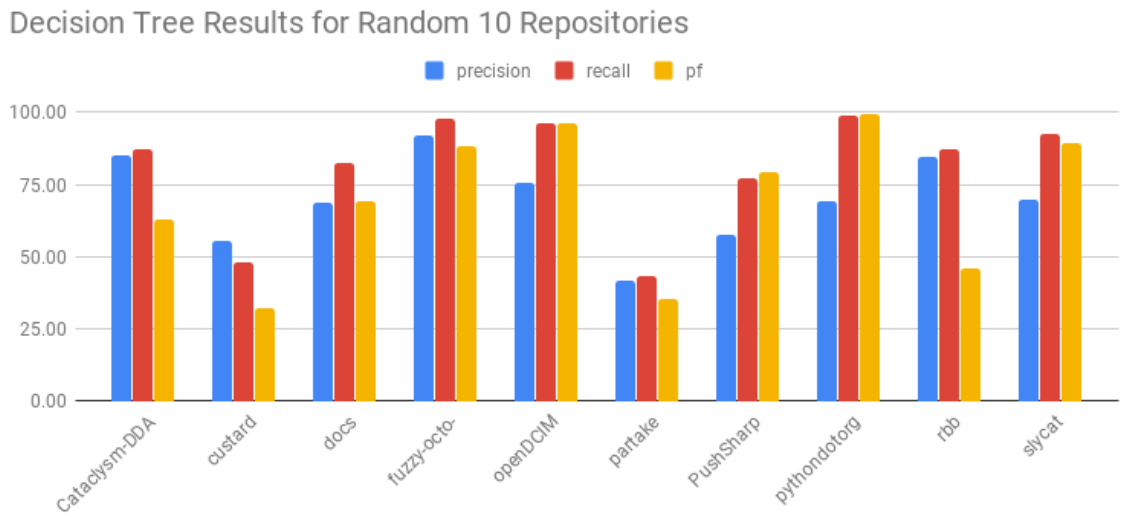

Figure 5: Averages for pf, precision and recall for all time classes on Random 10 repositories

Here we also see how the amount of data fed to the classifier affects the prediction. Especially for the *pf* value, we see these repositories do not produce results that are as good as the ones from the top 10 repositories. Figure 6 shows the distribution of time class of repositories in Random 10 dataset.

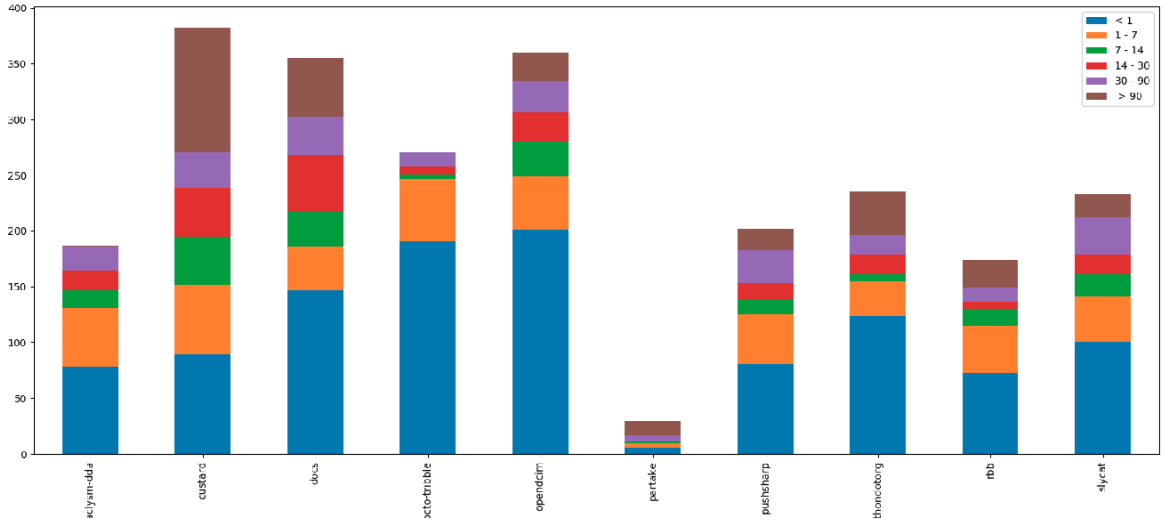

Figure 6: Time class distribution of the issues of each Random 10 repository

4.1.3 Experiment 3: Cross Validation on *combinedMenzies*, *combinedRandom10, combinedTop10 and combinedAll* Datasets

Table 6 holds the results from this experiment. CombinedTop10 has the best precision overall, excluding the time class 1 case. However, it still has large pf values, compared to combinedMenzies dataset, which has very low pf values and fairly good precision values.

		Cross Validation			Issue Counts	
		precision	**recall**	**pf**	**Positive**	**Negative**
combinedMenzies	1	51	15	3	7279	40234
	7	66	63	11	12173	35340
	14	71	64	11	13920	33593
	30	75	75	13	16132	31381
	90	79	72	15	21351	26162
combinedTop10	1	64	52	27	12269	13310
	7	71	91	70	16689	8890
	14	75	99	87	18518	7061
	30	84	97	81	20822	4757
	90	92	100	100	23638	1941
combinedRandom10	1	68	11	4	1088	1318
	7	65	90	84	1511	895
	14	71	100	100	1692	714
	30	80	100	100	1893	513
	90	89	100	100	2122	284
combinedAll	1	54	21	14	275168	366357
	7	63	95	87	389141	252384
	14	71	96	88	442404	199121
	30	78	100	100	500669	140856
	90	90	100	100	574671	66854

Table 6: Cross validation results for decision trees on combined repositories.

On top of the cross validation experiment, we also gathered the results from experiment #1 and experiment #2, calculates insight values such as mean, standard deviation and minimum and maximum values, for each columns. These values can be seen from Table 7. This

allowed us to see whether if the combined datasets perform better or worse than the average.

These average values are calculated from the previous experiments. The purpose of this is to check whether combining the data before the execution yields to better results than training several models for each dataset and taking their average values. Of course, getting the averages for the *combinedAll* case was not possible, since it contained more than 4.000 repositories and it would be impossible to train modes for each of them.

		Average Cross Validation								
		precision			recall			pf		
		mean	stdev	min.. max	mean	stdev	min.. max	mean	stdev	min.. max
combinedMenzies	1	32.8	34.9	0..78	25.6	29.5	0..77	10.2	14.1	0..41
	7	43.1	37.7	0..80	48.4	44.6	0..100	25.3	33.6	0..89
	14	50.5	36.4	0..86	56.9	43.6	0..100	22.7	30.6	0..83
	30	54	38.9	0..91	66	45.9	0..100	20.4	29.9	0..77
	90	74.4	21.6	32..96	73.7	37.1	2..100	14.1	20.3	0..71
combinedTop10	1	68.4	9.4	56..83	51	34.2	1..95	22	18.9	0..57
	7	73.9	12.9	59..91	92.9	55.8	85..100	62.1	30.3	13..100
	14	78.7	12.3	62..94	96.5	4.3	87..100	64.5	34.5	22..100
	30	83.8	9.8	67..96	98	2.2	93..100	72.2	29.4	25..100
	90	92.4	6.6	80..100	99.3	1.4	96..100	91.2	16.6	53..100
combinedRandom10	1	51.4	31.5	0..100	61	34.9	0..100	39.3	36.4	0..100
	7	62	14.2	47..93	80.7	27.9	17..100	65.5	36	6..100
	14	68.5	26	0..94	82.2	30.1	0..100	75.1	28.3	25..100
	30	74.2	27.2	0..97	82.8	31.8	0..100	76.7	32	25..100
	90	88.3	11.6	61..100	98.7	2.5	92..100	82.4	31.4	0..100

Table 7: Average values for precision, recall and pf from the experiments on individual repositories.

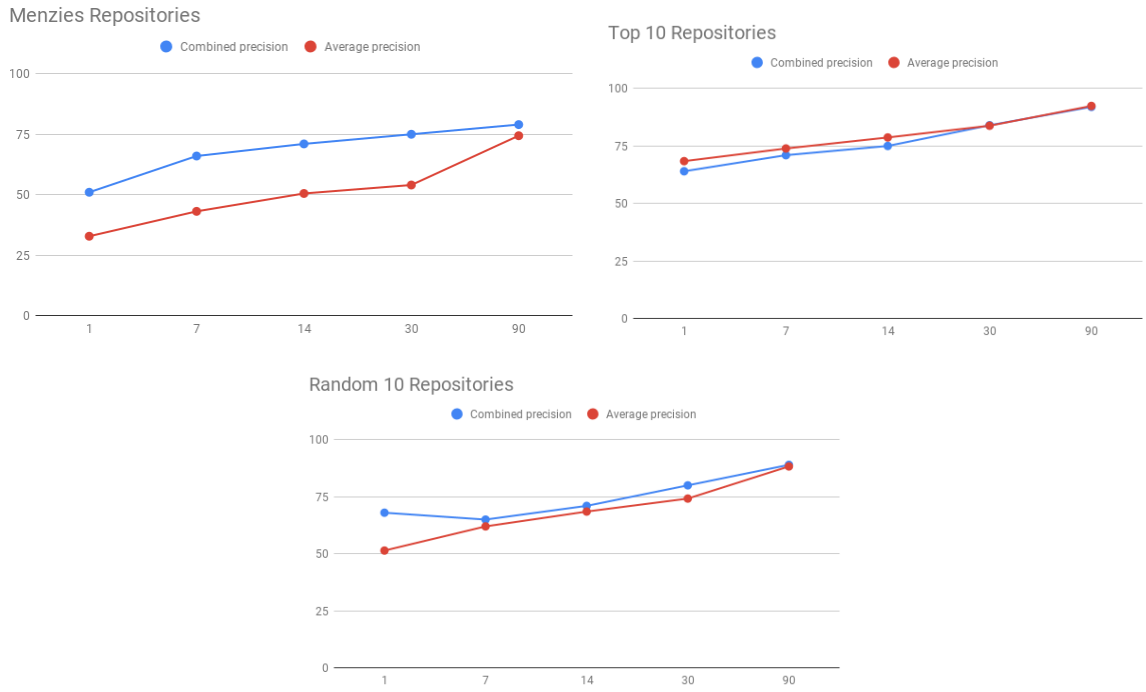

Figure 7: Comparison between the precisions from the combined datasets and the average values for precisions of the individual repositories.

Figure 7 shows a comparison of precision values between combined and average results. We see a fluctuating result between the combined precision values and average precision values of 10 repositories. In *Menzies* dataset, combined precision is almost twice the average precision, which can be explained by the different structures of the repositories. Some of the repositories within this dataset results in 0 precision due to their skewed time class distribution, which yields to very low average precision values. In the combined case however, the dataset contains *some* issues from time class 1, and the results are reflecting this change as well.

For the case of *random* and *top* repositories, since the individual time class distribution is somewhat similar between the repositories, we see that the precision trend is also similar between the combined and average cases. For *random* repositories, the one time class results seem to differ, which can be explained by the structure of the *partake* repository, which has almost zero issues from the aforementioned time class, which in turn results in very low precision for that experiment.

4.2 Deep Learning Implementation Results

Like the previous section, this one displays the results of experiments ran on the different datasets from Section 3.2. However, this section also includes an experiment on repositories of Menzies dataset, which we skipped in the previous section, since it was just the replication of the baseline, which was shown in Section 3.3.

4.2.1 Experiment 1: Cross Validation on *Top10* Dataset

Table 8 shows the results of this experiment. First thing to notice here is that compared to the decision trees results, neural networks have larger *pf* values. In the decision trees results, we never saw the *pf* value exceed the value for *recall* and mostly they were below the *precision* as well. However, in the case of the deep learning algorithm, we often see that the false alarms (*pf*), are more often than the correct classifications.

		Cross Validation		
		precision	**recall**	**pf**
appium	1	43	100	100
	7	57	86	88
	14	66	97	95
	30	0	0	0
	90	90	99	100
Font-Awesome	1	69	100	100
	7	76	24	23
	14	76	41	40
	30	83	49	22
	90	87	100	100
framework	1	52	31	17
	7	59	48	28
	14	68	90	72
	30	83	98	97
	90	94	100	100
hd	1	42	58	61
	7	68	98	32
	14	76	98	23
	30	86	98	90
	90	97	100	70
ionic	1	35	34	29
	7	46	33	0
	14	69	99	100
	30	81	93	92
	90	94	100	100
ppsspp	1	50	71	70
	7	58	100	100
	14	63	89	88
	30	70	81	76
	90	81	93	83

RedisDesktopManager	1	17	74	61
	7	50	91	85
	14	66	94	83
	30	87	99	89
	90	99	100	40
SCII-External-Maphack	1	57	88	88
	7	77	100	80
	14	81	100	80
	30	89	100	60
	90	99	100	40
steam-for-linux	1	43	38	13
	7	64	100	100
	14	70	47	48
	30	82	56	50
	90	88	100	100
wet-boew	1	19	4	4
	7	61	98	98
	14	67	96	97
	30	75	100	100
	90	78	88	98

Table 8: Cross validation results for deep learning on Top10 repositories.

Having a small amount of issues from the negative class, or having a substantially large amount of issues from one issue type results in an almost 100% *recall*. The averages of the results from Table 8 can be seen on Figure 8.

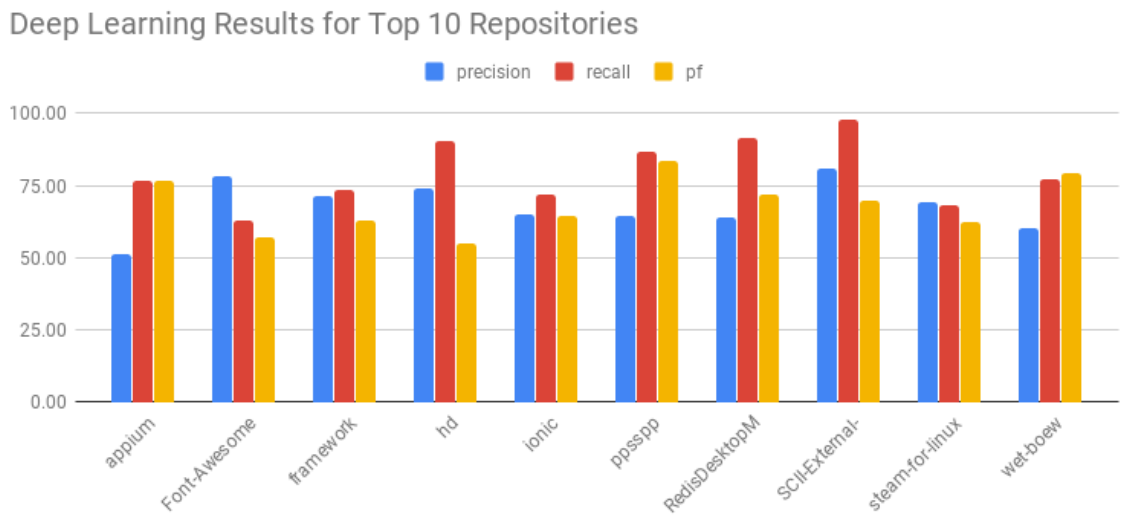

Figure 8: Averages for pf, precision and recall for all time classes on Top 10 repositories

In general, the neural network produces good results in terms of *precision* and *percentage.* However, the large numbers under the *pf* column makes these predictions unreliable.

Table 9 shows the time class distribution on issues for this dataset.

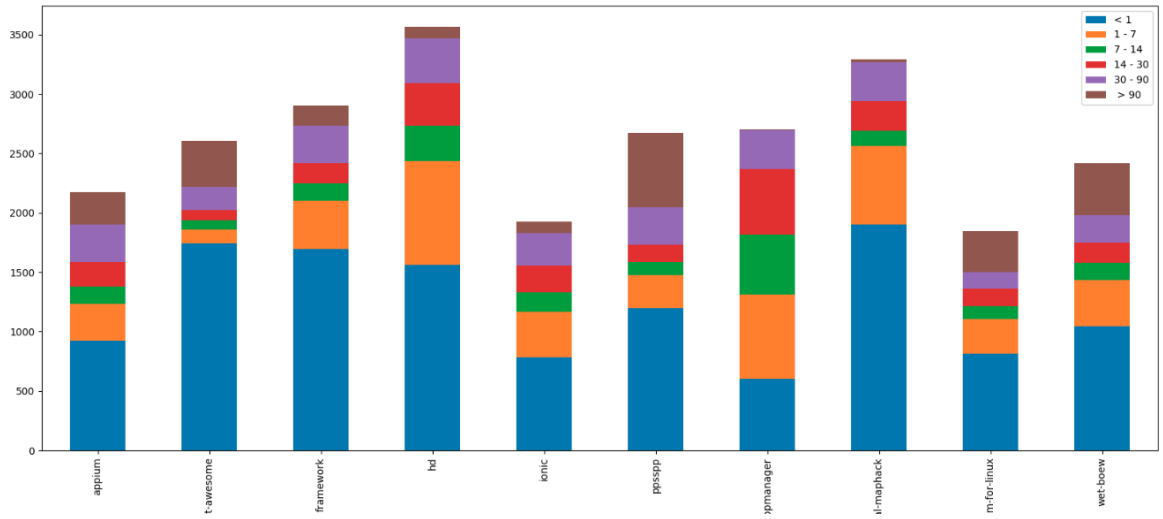

Figure 9: Time class distribution of the issues of each Top 10 repository

4.2.2 Experiment 2: Cross Validation on *Random10* Dataset

The results are shown in Table 9. Like the decision trees algorithm, we see how the lack of data is affecting the performance of the classifier.

		Cross Validation		
		precision	**recall**	**pf**
Cataclysm-DDA	1	1	3	4
	7	65	29	37
	14	78	100	90
	30	89	76	49
	90	99	100	10
custard	1	7	6	7
	7	36	77	71
	14	50	91	92
	30	62	94	94
	90	70	100	90
docs	1	41	24	17
	7	33	27	28
	14	57	49	50
	30	78	91	82
	90	89	93	62
fuzzy-octo-tribble	1	70	78	79
	7	90	96	70
	14	92	100	50
	30	95	97	20
	90	100	100	0
openDCIM	1	59	70	71
	7	69	78	77
	14	75	86	92
	30	84	100	100
	90	92	99	90
partake	1	18	25	73
	7	10	5	3
	14	30	35	68
	30	0	0	0
	90	61	90	68
PushSharp	1	29	21	19
	7	65	63	65
	14	70	79	76
	30	80	88	68
	90	90	75	54

pythondotorg	1	47	33	40
	7	65	86	84
	14	70	68	61
	30	76	92	88
	90	82	95	98
rbb	1	42	82	80
	7	68	80	65
	14	79	86	56
	30	81	100	70
	90	89	91	57
slycat	1	45	30	26
	7	61	32	35
	14	72	66	58
	30	79	77	77
	90	93	85	36

Table 9: Cross validation results for deep learning on Random10 repositories.

Figure 10 displays the averages for all three metrics. Overall, *prediction* and *recall* are quite low for these repositories. Especially with the repository *partake,* the full effect shows.

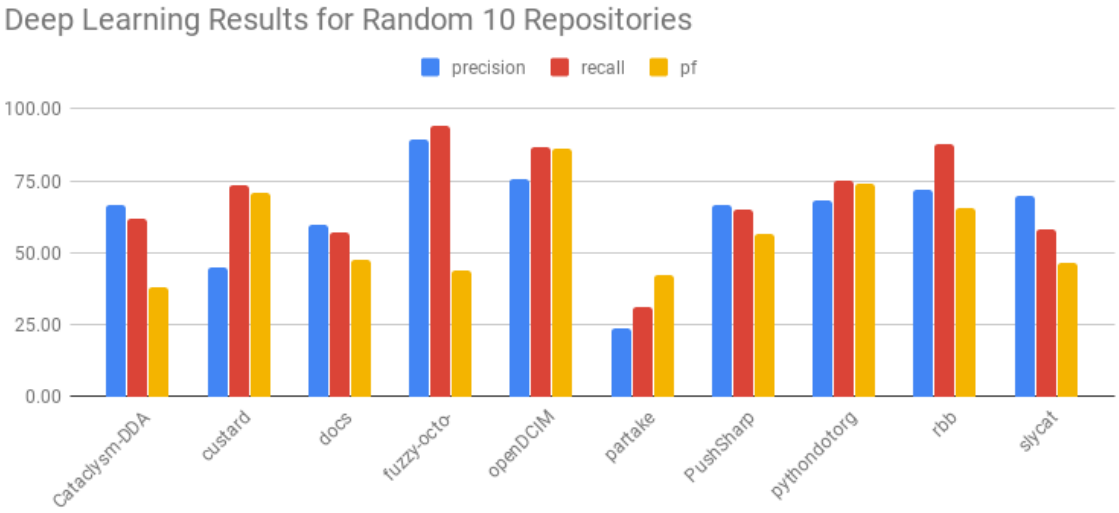

Figure 10: Averages for pf, precision and recall for all time classes on Random 10 repositories

The repository has around 100 issues in total, and that yields to a *precision* of 24 on average. Figure 11 shows the distribution of issue classes of each repository in this dataset.

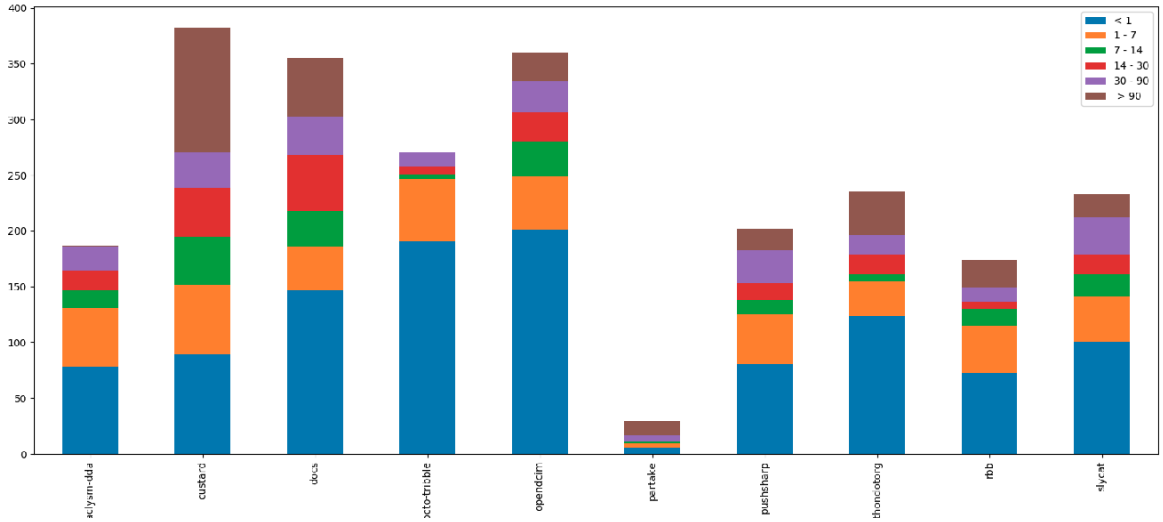

Figure 11: Time class distribution of the issue issues of each Random 10 repository

4.2.3 Experiment 3: Cross Validation on *Menzies* Dataset

Menzies dataset has the worst performance among all datasets. The results are shown in Table 10. Even the random repositories outperform Menzies dataset, even though their average size is almost 5 % of Menzies'.

		Cross Validation		
		precision	**recall**	**pf**
camel	1	32	43	0
	7	36	49	0
	14	37	84	21
	30	68	96	9
	90	73	98	14
cloudstack	1	0	0	0
	7	71	85	85
	14	78	90	92
	30	85	99	100
	90	93	100	100
cocoon	1	0	0	1
	7	13	7	5
	14	12	10	8
	30	19	14	12
	90	19	20	17

deeplearning	1	0	0	0
	7	79	95	97
	14	85	100	100
	30	90	99	100
	90	94	99	100
hadoop	1	0	0	0
	7	0	1	0
	14	0	0	0
	30	0	0	0
	90	0	0	0
hive	1	0	0	0
	7	0	0	0
	14	0	0	0
	30	0	0	0
	90	26	7	8
kafka	1	38	44	44
	7	64	90	90
	14	72	93	92
	30	80	99	99
	90	89	100	100
node	1	0	0	0
	7	68	100	100
	14	76	98	99
	30	83	99	100
	90	88	98	100
ofbiz	1	4	1	1
	7	36	17	16
	14	50	42	43
	30	58	84	85
	90	68	99	99
qpid	1	0	0	0
	7	0	0	0
	14	0	0	0
	30	25	0	0
	90	21	3	3

Table 10: Cross validation results for deep learning on Menzies repositories.

Figure 12 shows the averages for precision, recall and pf for each repository in this dataset. The reason why some of the repositories have bad results is the issue class distribution of these repositories. Other datasets have most of their issues closed under under 90 days, but repositories in this one have mostly old issues. Therefore upon training the classifier, it fails to recognise any issues that were closed in less than 1 or 7 days, therefore the precision value turns out quite low.

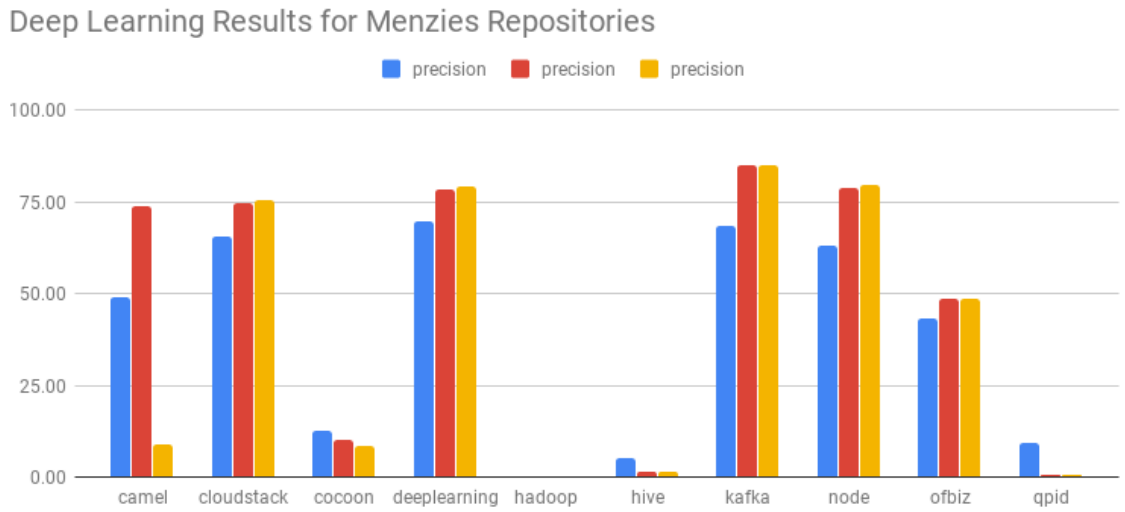

Figure 12: Averages for pf, precision and recall for all time classes on Menzies repositories

This concept is especially visible in the results of *qpid, hive* and *hadoop* repositories. The rest seems to be producing normal results since the issues time classes are more evenly distributed. This distribution of issue time classes are shown in Figure 13. If we compare the results between *camel* and *hive* repositories, they are quite different even though the repositories have similar amounts of issues in total. And on top of this, the number of issues that are closed after 90 days is similar as well. However, *camel* has some amount of issues that were closed within a day or a week, where *hive* lacks these kinds of issues. This property yields into two completely different results for these two repositories.

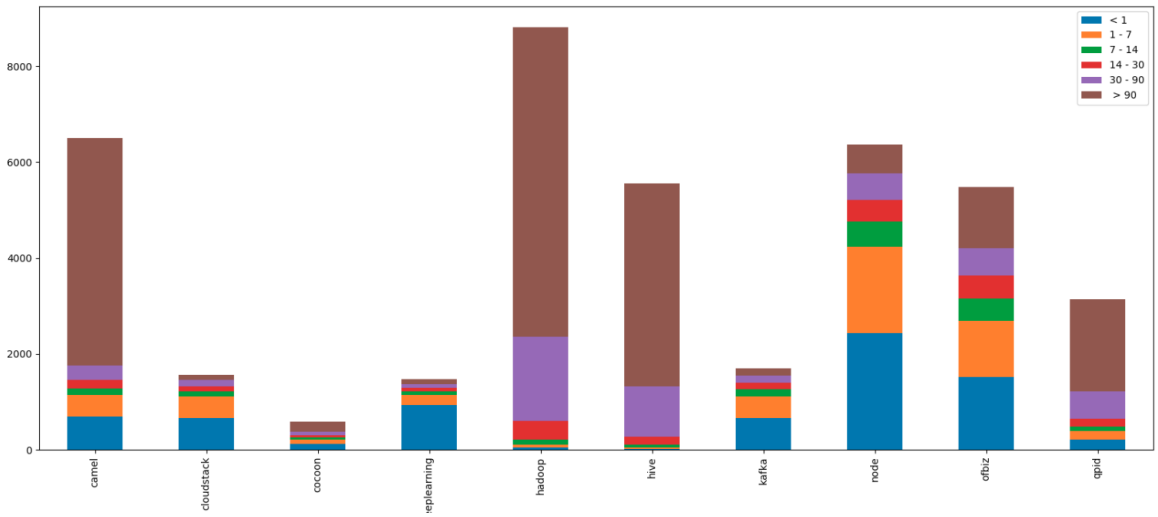

Figure 13: Time class distribution of the issue issues of each Menzies repository

4.2.4 Experiment 4: Cross Validation on *combinedMenzies*, *combinedRandom10*, *combinedTop10 and combinedAll* Datasets

Table 11 shows the results for this experiment. We see combinedMenzies has poor performance compared to the rest of the datasets. CombinedRandom10 dataset has great results, compared to its small size. However, the large numbers under pf column shows that predictions coming from these classifiers are biased.

		Cross Validation		
		precision	**recall**	**pf**
combinedMenzies	1	22	3	4
	7	26	18	16
	14	29	25	23
	30	34	25	24
	90	45	33	32
combinedTop10	1	37	44	24
	7	63	76	74
	14	70	91	88
	30	78	96	96
	90	88	99	100
combinedRandom10	1	43	11	12
	7	65	92	92
	14	72	97	96
	30	81	100	100
	90	92	100	100

	1	58	43	19
	7	60	88	88
combinedAll	14	68	96	96
	30	78	100	100
	90	92	100	100

Table 11: Cross validation results for deep learning on combined repositories.

5. Discussion of the Results

Menzies et al. argued that simple, human readable classifiers such as Decision Trees performed better in the context of predicting an issues lifetime. In this paper we applied one of the most complicated classifiers of today to this problem. After running the aforementioned experiments, the results we received showed us that a complex classifier such as a deep neural network can perform well under right circumstances, however it is heavily reliant on the shape and the size of the data.

The performance of both algorithms are quite different between *Menzies dataset* and the rest of the datasets. This can easily be explained by looking at each datasets class distribution. Menzies dataset has nearly half of its issues closed after the 90 days threshold, whereas the rest of the datasets have around 10% of their issues closed after 90 days period. When training, this leads to very biased training data. For instance, for the *combinedTop10* case, when training the 90 days classifier, %91.4 of the issues in the training data were closed before 90 days. This problem results in three things: high precision, high recall and high false alarms.

While training, since the classifier is exposed to instances from one class most of the time, it predicts that class almost all the time. The effects of these can be seen in the 7, 30 and 90 classifiers for *combinedAll, combinedTop10* and *combinedRandom10* datasets. The *recall* for these 9 classifiers averages *97.6* in the case of the DNN experiment, which shows how biased the neural network is. This also explains the 100 cases for the *recall* and *pf* values. When the classifier gets this biased and predicts everything will be the 0 case, that is, predicting every issue to be closed within the given days, it predicts all the cases where the issue was actually closed before given days correctly, and fails to predict the correct class of all of the remaining issues.

For all datasets, the neural network approach yields in an almost one to one relationship between the *percentage of issues that were closed before the time class* and the *precision*

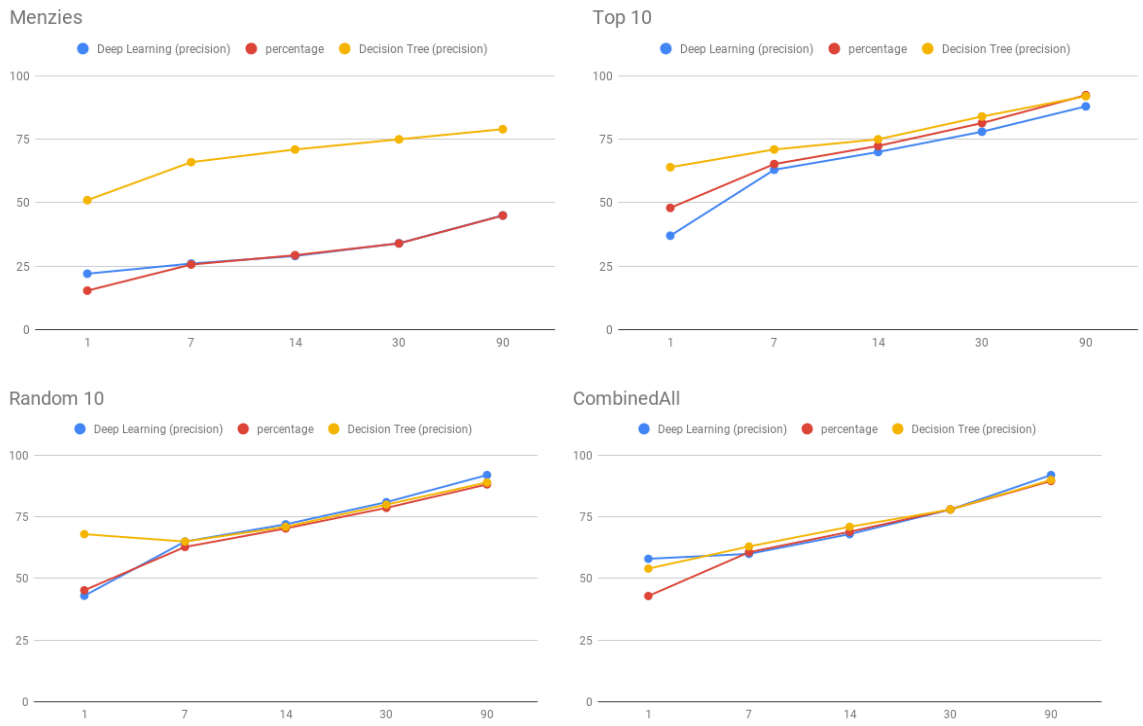

Figure 14: Precision values for deep learning and decision trees compared to the percentage of issues for each time class.

value of the results. Looking at the graphs, one can easily see the link between the two values for the combined datasets.

For the case of decision trees however, we see some amount of differentiation between the two values. Especially for the *Menzies* dataset, the precision results seems to be much better than the amount of issues used to train the model does not seem to affect the precision result that much.

Overall, the decision trees algorithm seems to outperform the neural network implementation in terms of precision. Neural network models tend to get really biased when the training data contains samples from one specific target class.
In general, we see that neural networks are heavily reliant on the data and their distribution. Decision trees while relying on the dataset distribution, provides slightly better results. For instance, for *Menzies* dataset, decision tree classifier outperforms neural networks by a great margin. For the case of time class 1 classifier, the margin is more than 100 %. Precision of the deep learning algorithm is 22 and precision of decision trees is 51. Getting a 0.51 precision where 17.68 % of the data belongs to the target class is a big success.

The different distributions of the features could have also lead to the different results in Figure 14. If the values for a given feature is very diverse, it might lead to good predictions. However, a feature with very small standard deviation would have very little effect on the prediction. Top10, Random10 and CombinedAll datasets all yield in similar results for both algorithms. However for Menzies dataset, which comes from a separate distribution, the results vary greatly.

Neural networks also excel with the amount of features a dataset has[15]. Our dataset contained only 7 features, where all of them were *integer* features. This shortage of features also explains the poor performance of our models. Simple classifiers such as decision trees however, excel with small amount of distinct features.

Deep neural networks, or neural networks in general also perform much better when they are exposed to large datasets. However, in our experiments we found that this is not always the case. Looking at the performances of the *Menzies* and *combinedRandom10* datasets, we see that the size of the dataset did not have a big effect of the performance of the classifiers. On the contrary, *combinedRandom10* datasets deep neural network performance is much better than Menzies dataset's, even though the latter dataset has roughly 20 times more issues than the former.

6. Prediction Tool

During the time we were conducting the experiments with deep learning, we decided to implement a prototype tool which a user could easily use to predict how long it would take to close any given issue. Since version control systems are widely used in today's software development, we think that the best implementation for this would be to have a plugin for any version controlling system, where upon creation of the issue, the plugin would ask the *previously trained models* to predict the lifetime of the issue. The models would return an interval, such as *between 7 and 14 days, before 1 day* or *after 90 days*.

The plugin then would have to handle the preprocessing of the data, training the model and updating the model in the background. Updating the model would be a periodic task where in a given schedule, the plugin would retrain the model with the newly added data, to make the prediction models a bit more reliable by making them exposed to more data.

As the first prototype, we implemented a web-based application, where users are able to select the dataset they want to experiment with. At the moment, the choices are limited to the combined datasets, namely combinedM*enzies, combinedTop10, combinedRandom10* and *combinedAll*. However, extending this to any given dataset is fairly simple, since the preprocessing tools provided by Menzies et al. are quite useful and reusable.

Figure 15 shows the homepage of our prediction tool. In this screen of the app, users are prompted with the dataset that they would like to use, and regarding their choice, the *pre-trained* models of the selected dataset are loaded to the memory of the application. After

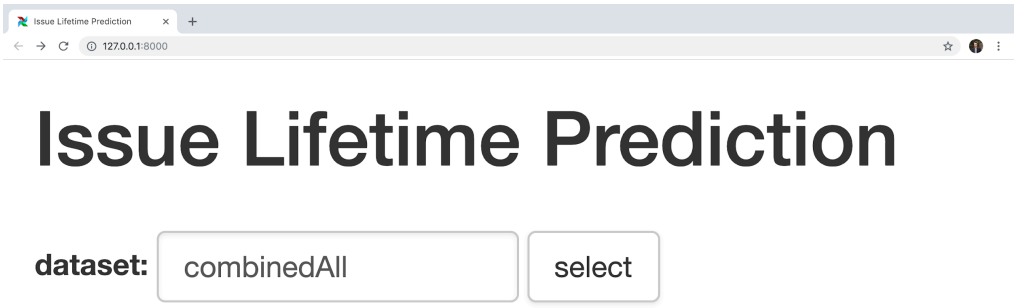

Figure 15: Homepage of the prediction tool

selecting the dataset, the users are forwarded to the page where they can input the values for the features and get a prediction based on the input values instantly.
In the second screen of the applications, the users are prompted to enter the values for the features for testing purposes. If we had access to the raw data, we could just make the user import an issue object in a specific format, such as *json* or *csv*. Then calculating the values for some features wouldn't be a hard task, since we could use a simple database, or even a dictionary, to store the mapping between *creator_id* and how many issues they created in general, or just in the project and so on. The same can easily be done for issues. Some features such as *nCommitsInProject* requires us to have access to whole dataset for us to be able to calculate them. Considering all these in mind, we went with the simple interface that can be seen in *Figure 16*. In this web page, users can enter any value for the features, and our application would convert it into the format the models would accept, then input the given features into the five different models, and return their results to the user.
For the data in *Figure 16*, we see an issue of 550 characters is given as an input. The project has 520 commits, where 92 of those belongs to the person who created the issue. Before this issue was created, the project had 420 issues, 365 of them were closed. Out of this 420 issues, 72 belonged to our creator and 65 of those were closed. After giving these values to five different models, which basically predict if the given issue will be closed before *1 day, 1 week, 2 weeks, 1 month* and *3 months*, respectively, denoted by the table on the bottom of the page.

The table gives us an interval, in which we predict the issue will be closed. Looking at the table, we can deduct that this issue will be closed somewhere between *7* and *14* days.

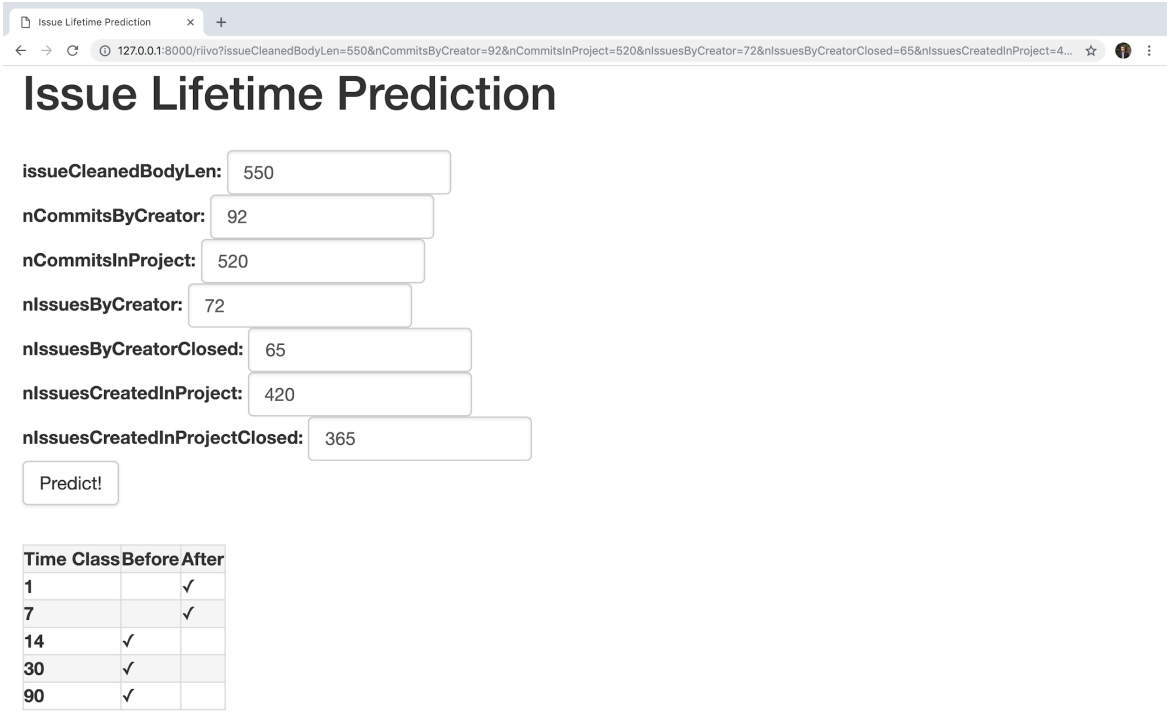

Figure 16: Interactive result page of the prediction tool

7. Conclusion

Predicting the lifetime of an issue is not easy. A lot of researchers approached this problem with different algorithms for the last 20 years. In this paper, we examined a possibility of having a better classifier for the problem of predicting an issues lifetime. Existing studies tried several different approaches, and we selected to use a deep learning approach to IRT problem.

We found that replication of an existing study under a different context is not always straightforward. We were not able to replicate some parts of the existing studies with the same code and same data. Often, the implementation does not match the results given in a study. One of our aim in this study was to make our implementation as readable and as reusable as possible for future researchers.

The paper posed two research questions.

1. How can one select and implement a deep learning algorithm to predict issue resolution time?

A simple feed-forward neural network was chosen to be our deep neural network algorithm. Dataset limitations, such as limited amount of features, no textual features and overall structure of the data led us to this decision. Using a popular Python framework called Keras, we were able to create a three level neural network, which had 7 nodes in its input layer, a 16 node hidden layer repeated twice and lastly, a single node output layer.

The decision of the class was made by looking at the prediction value, and determining the class by comparing it to 0.5.

2. How does a deep neural network perform against other conventional classification algorithms (i.e. Decision Trees and Random Forest)?

We chose a recent study which implements a decision tree algorithm to tackle the IRT prediction problem as the baseline. According to our experiments, the baseline algorithm has higher *precision* and *recall,* while having a lower value for the false alarms, *pf.* Looking at the *combined* datasets, our neural network implementation has an average precision of **60.05**, whereas the decision trees implementations average precision is **72.85.** For the case of *recall,* DNN implementation's average is **66.85** and the baseline average is **77.05.** For the failed cases, DNN has an average of **64.2** and where decision trees' average *pf* value is **59.75.**

Overall, our deep learning implementation failed to meet the performance of the baseline algorithm in most aspects. We believe the lack of features and lack of diversity in them yielded in these results.

8. References

1. Kikas, Riivo, Marlon Dumas, and Dietmar Pfahl. "Using dynamic and contextual features to predict issue lifetime in GitHub projects." *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016.
2. Panjer, Lucas D. "Predicting eclipse bug lifetimes." Proceedings of the Fourth International Workshop on mining software repositories. IEEE Computer Society, 2007.
3. Bhattacharya, Pamela, and Iulian Neamtiu. "Bug-fix time prediction models: can we do better?." *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 2011.
4. Guo, Philip J., et al. "Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows." *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010.
5. Hooimeijer, Pieter, and Westley Weimer. "Modeling bug report quality." Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. ACM, 2007.
6. Raja, Uzma. "All complaints are not created equal: text analysis of open source software defect reports." *Empirical Software Engineering* 18.1 (2013): 117-138.
7. Gousios, Georgios, and Diomidis Spinellis. "GHTorrent: GitHub's data from a firehose." *Mining software repositories (msr), 2012 9th ieee working conference on*. IEEE, 2012.
8. Breiman, Leo. "Random forests." *Machine learning* 45.1 (2001): 5-32.
9. Rees-Jones, Mitch, Matthew Martin, and Tim Menzies. "Better predictors for issue lifetime." *arXiv preprint arXiv:1702.07735* (2017).
10. Hall, Mark Andrew. "Correlation-based feature selection for machine learning." (1999).
11. Gulli, Antonio, and Sujit Pal. *Deep Learning with Keras*. Packt Publishing Ltd, 2017.
12. Abadi, Martín, et al. "Tensorflow: a system for large-scale machine learning." *OSDI*. Vol. 16. 2016.
13. Bergstra, James, et al. "Theano: A CPU and GPU math compiler in Python." *Proc. 9th Python in Science Conf*. Vol. 1. 2010.

14. Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980*(2014).
15. Verikas, Antanas, and Marija Bacauskiene. "Feature selection with neural networks." Pattern Recognition Letters 23.11 (2002): 1323-1335.

Appendix

I. Code

The code used in this study can be found at: https://github.com/atakanarikan/issue-LifetimePrediction

II. License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Atakan Arıkan**

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

Issue Report Resolution Time Prediction Using Deep Learning Techniques

supervised by **Dietmar Pfahl**.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Atakan Arıkan
14/02/2019