Categorizing Software Defects using Machine Learning

Viktor Stagge
Categorizing Software Defects using Machine Learning

Viktor Stagge
ine12vst@student.lu.se

May 18, 2018

Master’s thesis work carried out at Sony Mobile Communications AB.

Supervisors: Kenneth Ulrich, kenheth.ulrich@sony.com
Markus Borg, markus.borg@ri.se

Examiner: Jacek Malec, jacek.malec@cs.lth.se
Abstract

We analyze how automatically generated crash reports can be used to aid in the process of software defect categorization. The crash reports are automatically generated logs, which vary widely in both format and in information content. Each crash report used is linked to its corresponding, human-written bug report.

The problem is handled as a long text-based classification problem. Several different machine learning techniques are compared. Amongst these is our own Keras-based implementation of a Hierarchical Attention Network, with which we achieved an accuracy of 72.5% on Severity prediction, and an accuracy of 51.4% on Responsible Group prediction.

Keywords: crash reports, software defects, text classification, hierarchical attention networks, machine learning
Acknowledgements

We would like to thank everyone included from Sony Mobile Communications side. In particular, special thanks to Kenneth Ulrich, Johan Roos and Victoria Welbert for their guidance and help with this report and project.

I would also personally like to thank Thore Husfeldt for being a great source of both inspiration and knowledge during my time at Lund University.
# Contents

1 Introduction ........................................... 7  
   1.1 Background ........................................... 7  
   1.2 Related Work ......................................... 8  
   1.3 Contribution Statement .............................. 8  

2 Project Description ..................................... 9  

3 Theory .................................................. 11  
   3.1 Deep Neural Networks ............................... 11  
      3.1.1 Multi Layer Perceptron .......................... 11  
      3.1.2 Loss Function ................................... 12  
      3.1.3 Backpropagation .................................. 12  
      3.1.4 Activation functions ............................ 12  
      3.1.5 Input ............................................. 12  
      3.1.6 Dropout ........................................... 13  
   3.2 Convolutional Neural Networks .................... 14  
      3.2.1 Stride ............................................. 14  
      3.2.2 Pooling .......................................... 14  
      3.2.3 Dilation .......................................... 15  
   3.3 Recurrent Neural Networks ......................... 15  
      3.3.1 LSTM ............................................. 15  
      3.3.2 GRU .............................................. 16  
      3.3.3 Stateful Recurrent Neural Networks ............ 16  
   3.4 Hierarchical Attention Networks .................. 16  
      3.4.1 Word Encoder ..................................... 16  
      3.4.2 Word Attention .................................... 16  
      3.4.3 Sentence Encoder ................................. 17  
      3.4.4 Sentence Attention ............................... 17  
      3.4.5 Classification .................................... 18  

Chapter 1
Introduction

Software deployed at a large scale generates a large number of crashes and crash reports. Attempting to categorize such data by hand or via expert systems can be time-consuming and is prone to error. During recent years the applications and effectiveness of machine learning techniques have vastly increased. Machine learning offers new ways of efficiently finding connections within large quantities of data. In this report we focus on classifying large text files containing automatically generated crash reports. The predictability of two kinds of crash categorization, Severity and Responsible Group, are explored.

1.1 Background

The introduction of bugs in a software system is almost inevitable. In order to have an appropriately functioning software system, a great deal of time and effort goes into finding and fixing the underlying problems which cause bugs. Any form of automation that can ease this problem would reduce the amount of work hours needed to find a bug, allowing more bugs to be found, and resulting in an increase in the overall quality of the software system.

Sony Mobile have for several years collected automatically generated crash reports which contain logged events that occurred around the time of the crash. The resolved crash reports have been manually labeled, enabling a supervised machine learning approach.

There have been many attempts to use text from bug reports to yield a classification that can ease the process of correcting found bugs. A large amount of this work has been done on Severity prediction [15] [5] [11], using the text from manually written bug reports. There have also been multiple attempts to predict which personnel may be capable of resolving a given bug [18] [3], this also using the text in the manually written bug reports. Furthermore, not all reported bugs are actually bugs, and there have been several attempts to classify whether or not a manually reported bug actually is a bug [24] [1] [2].

In this report we focus on automatically generated crash reports. The crash reports are
in a text format, and consist of information from multiple sources in both software and hardware. This may include certain types of built-in logs, as well as the memory states in the affected device. The crash reports are processed and then used as input to several machine learning techniques to classify a known field for the crash report, such as Severity and Responsible Group.

We found that the superior model for the text based classification of automatically generated crash reports is Hierarchical Attention Networks [23]. Using this we achieved a 72.5% for Severity prediction, and a 51.4% for Responsible Group prediction. The Hierarchical Attention Network was implemented in Keras [7], with the attention mechanism implemented as a Keras Layer, allowing the modularity that an arbitrary Keras Layer provides. In this paper, this attention mechanism was used on both a word-by-word and a line-by-line basis.

1.2 Related Work

Most of the related work applies different forms of natural language processing techniques to text from a manually written bug report. Menzies et al. [15] tried to generate rules based on tokenized and processed bug report texts, in order to predict the respective bug report’s Severity. Others have attempted to use several other machine learning techniques, such as Naïve Bayes, k-Nearest Neighbours, and Support Vector Machines in order to predict the bug report’s Severity [5] [11].

Closely related to our attempts of finding a suitable Responsible Group from our automatically generated crash reports are previous attempts of finding suitable personnel [3] from bug reports. Anvik et al. explored the possibilities of applying machine learning techniques on manually written bug reports in order to find a suitable person who could fix the bug. The classifications were in their study made using Naïve Bayes, SVM, and decision trees.

Furthermore, the attempts of classifying bug reports as either bug or non-bug [24] [1] [2] adheres to the same form of text-based classification problem of bug reports as the similar work on Severity prediction.

1.3 Contribution Statement

This report shows interesting tendencies regarding the potential connections that could be found between automatically generated crash reports and their respective, underlying bug.
Chapter 2
Project Description

When a crash occurs on a Sony Mobile test phone, a crash report is automatically logged and sent to a server. These crash reports are then used to form bug reports, which are concise reports of the issues causing the crash, or group of crashes.

Since an underlying bug tends to yield multiple crashes across different phones and platforms, there is a need for grouping crashes which are practically the same. This is done using a semi-automated system that utilizes expert users. Grouping the crashes requires extensive domain knowledge and fundamental understanding of the crash reports, as well as a substantial amount of time. A brief summary of the current system can be seen in figure 2.1.

![Diagram of current system](image)

**Figure 2.1:** Current system for categorizing software defects at Sony Mobile Communications.

The main idea of this project is to evaluate the feasibility of devising a machine learning model which can aid the process of categorizing software defects. This is carried out by
using the fields from the large quantities of closed bug reports as target values, and the linked crash reports as input values.

The crash reports are long text files typically spanning over hundreds or thousands of lines. There are typically a few meta fields available, but these are present in less than 50% of the crash reports, and are rarely specific enough to provide any particular insight into the underlying issue.

To explore what kind of connections a machine learning model could potentially learn and use, while reducing specificity and the need for domain knowledge, the problem is handled as a long text classification problem. A specified selection of each crash report is used to predict the targeted value in its linked bug report.

Since only a limited amount of characters or words (also referred to as tokens) could be used as input to an arbitrary neural network, a quite substantial amount of most logs had to be removed.

Two fields from the bug reports have been selected as appropriate and feasible. These are **Severity** and **Responsible Group**. **Severity** is a 1 to 4 scale indicating how severe the bug is, while **Responsible Group** indicates which internal or external group that ought to be held responsible for fixing the underlying bug.

Although a near-perfect result would be needed in order for one to rely wholly on an automated system, a fairly precise result of **Severity** would potentially be sufficient to provide some preliminary ranking of the crash reports, so that the crash reports could be handled in an appropriate order. A fairly precise classification of the **Responsible Group** could drastically decrease the amount of potentially relevant crash reports during identification of similar types of crashes.
Chapter 3
Theory

3.1 Deep Neural Networks

A neural network is a structure which allows for the learning of an arbitrary mapping between a specified form of input, and a specified form of output [8]. The precision of this mapping is highly dependent on how the problem is represented and how the neural network itself is structured.

3.1.1 Multi Layer Perceptron

The fundamental component of a neural network is the Multi Layer Perceptron, MLP. A single layer has \(n\) nodes. Each node has a fixed size input, and a scalar output.

Each node’s output is calculated as a weighted sum of its inputs [8]. This weighted sum is then passed through a non-linear activation function, in order to yield the final output of that specific node.

![Figure 3.1: A Multi Layer Perceptron with inputs \(x_i\) and outputs \(y_j\).](image-url)
3. Theory

The weights which are used to calculate the weighted sum are specific for each node. An example of the MLP structure can be seen in figure 3.1

3.1.2 Loss Function

As a measurement of how correct, or incorrect, the mapping of a neural network is, a loss function is used [8]. This loss function gives a larger value when the network has supposedly performed worse. The network’s weights are then optimized such that the network yields an as small loss as possible. This is done through gradient descent, with backpropagation.

3.1.3 Backpropagation

To learn the weights during training, a minimization of the loss function is attempted through gradient descent [8]. The gradients are calculated for each layer through backpropagation, i.e. the gradients are calculated for each layer, starting at the output (using the loss function) and moving backwards through the network. The weights are then updated with a small step in the direction of the gradient.

3.1.4 Activation functions

There are a multitude of different non-linear activation functions that can be used in order to calculate the final output of a node in a neural network. Typically, the final layer for a classification task uses a softmax activation function [8]:

\[ \sigma(x)_j = \frac{\exp(x_j)}{\sum_i \exp(x_i)} \]

which acts as a "soft" maximum function, such that the largest value will become larger in comparison to the other values. A nice property of the softmax function is that the outputs will always sum to 1, and can be regarded as probabilities.

Other activation functions which are typically used inside the hidden layers of a neural network are the tanh and sigmoid activation functions. These can be regarded as smoother variants of the step-function. For convolutional neural networks, a relu activation function is typically used. The output of these activation functions can be seen in figure 3.2.

3.1.5 Input

Inputing text to a network can be done in multiple ways. The most straightforward way is to input each word, or character, as a one-hot encoded vector. This is a vector with one element for each possible word, where each index represents a word. For any given word, that index has the value 1. All other indices have the value 0. Typically, one index is reserved for unknown words.
3.1 Deep Neural Networks

Figure 3.2: The output for three common activation functions: sigmoid, tanh and relu.

Word Embeddings

Using a large vocabulary of one-hot encodings results in a large input dimensionality. For example, a vocabulary of 10,000 words yields an input dimension of 10,001, including a dimension for unknown words. This may become unfeasible, as certain words may only be seen a few times, and its corresponding weights may not have been updated a sufficient amount of times.

To reduce the input dimensionality, a word embedding can be used. Each word is then described as a k-dimensional vector [14]. Where k is typically on the order of 150, or 300. By letting this embedding be seen as any other trainable layer, the word embeddings can be learnt during training through backpropagation.

For each example during training, the weights from the embedding layer to the next layer will be updated. In comparison, when a one-hot encoding is used, only the weights for the occurred word will be updated. Hence, if the embedding layer tends to put somewhat similar words near each other in the k-dimensional space, they will benefit from the training of each respective example.

3.1.6 Dropout

A large issue when training neural networks is the concept of overfitting [22]. A practically proven approach in order to reduce the amount of overfitting is dropout [21]. Each node from a given layer in the neural network is then discarded with a probability $p_{drop}$, and kept with the probability $1 - p_{drop}$. During backpropagation, only the weights that contributed to the result are updated. I.e. only the weights for the nodes that were decided to be kept will be updated.

Practically, this can be seen as an ensemble of networks that utilize an extremely high level of weight sharing [21]. Hence, dropout contributes to the robustness of a model.
3.2 Convolutional Neural Networks

![Diagram of a Convolutional Neural Network](image)

**Figure 3.3**: A Convolutional Neural Network applied on text, at a word-by-word basis.

A convolutional network applies several filters of a fixed kernel size, $k$, onto the input. Each filter can be seen as a weighted sum of the $k$ input tokens, which is applied as a sliding window over the input [12]. The output for a filter at any given position is calculated by applying the convolutional layer’s activation function to the weighted sum. The weights of the convolutional filters can be initialized at random, and learnt during training through backpropagation.

### 3.2.1 Stride

The sliding window is moved along the input with a given step size, referred to as **stride** [20]. A stride of 1 implies that the sliding window is moved to include the immediately next input. For any filter size, $k$, above 1, this results in an overlap. The convolutional method used can be seen in figure 3.3.

### 3.2.2 Pooling

Pooling layers apply a given operation along the input, in order to subsample it[20]. This can be used on the output of a convolutional layer in order to reduce its dimensionality. **Max Pooling** [20] takes the maximum value within its filter size, $k_{\text{maxpool}}$, on the input. This is applied as a sliding window, where the stride is typically set to be the same as $k_{\text{maxpool}}$, as the idea is to reduce the locality of the result from applying the convolutional filters to the convolutional layer’s input.

The pooling layer can be applied to all of its input, essentially setting $k_{\text{maxpool}}$ to the same length as the length of the input to the pooling layer. This creates a **global pooling** [20] operation. Applying **global max pooling** yields the highest obtained value in the input,
which can be seen as corresponding to whether a specific filter has found a match anywhere in the input.

### 3.2.3 Dilation

By allowing gaps in the applied convolutional filters, it is possible to connect tokens which are located further away from each other. This, referred to as dilation [20], allows for broader connections. A dilation of 1 results in a filter which simply takes adjacent inputs. A dilation of 2 results in a filter which skips every second input token.

### 3.3 Recurrent Neural Networks

Recurrent Neural networks go over the input one token at a time, and for each token a temporary output is calculated. This output is then used as input together with the next token, in order to calculate the next temporary output. The final output is simply the temporary output which was calculated last.

The weight update can then be calculated by applying backpropagation at each step of the calculations mentioned above, starting with the last calculated temporary output [17]. This is referred to as backprogation through time.

#### 3.3.1 LSTM

Each Long Short-Term Memory cell, LSTM cell, has an internal state. For each input, the information stored in the internal state is updated by adding and subtracting information [9]. The next internal state is calculated according to [19]:

\[
\begin{align*}
    f_t &= \sigma(W_f \ast \bar{x}_t + b_f) \\
    i_t &= \sigma(W_i \ast \bar{x}_t + b_i) \\
    \tilde{C}_t &= \tanh(W_C \ast \bar{x}_t + b_C) \\
    C_t &= f_t \ast C_{t-1} + i_t \ast \tilde{C}_t
\end{align*}
\]

Where \( t \) is the index of the current token in the input. \( f_t \) is referred to as the forget gate, and \( i_t \) is referred to as the input gate, each being calculated at the index \( t \). \( W_f \) and \( W_i \) are the forget gates and input gates respective weight matrices, whilst \( b_f \) and \( b_i \) are the respective biases. \( \bar{x} \) is the input to each LSTM cell, which is comprised of a concatenation of the input token at index \( t \), and the output of the previous LSTM cell, \( h_{t-1} \). \( C_t \) is the internal state of the LSTM cell at index \( t \). The temporary output is, for each token in the input, calculated as:

\[
    h_t = \sigma(W_o \bar{x} + b_o) \ast \tanh(C_t)
\]

Where \( W_o \) is the weight matrix for the output, and \( b_o \) is the respective biases.
3. Theory

3.3.2 GRU

Gated Recurrent Unit cells, GRU cells, are a modified version of the LSTM cell [6]. The GRU cell combines the input and forget gate into one gate, referred to as the update gate. It also merges the internal state with its temporary output.

The temporary output for each step, \( t \), in the input sequence can be calculated according to:

\[
\begin{align*}
  z_t &= \sigma(W_z \ast \bar{x}_t) \\
  r_t &= \sigma(W_r \ast \bar{x}_t) \\
  \tilde{h}_t &= \tanh(W_h \ast \hat{x}_t) \\
  h_t &= (1 - z_t) \ast h_{t-1} + z_t \ast \tilde{h}_t
\end{align*}
\]

Where \( \bar{x}_t \) is the concatenation of the temporary output from the previous input token, \( h_{t-1} \), with the input, \( x_t \). \( \hat{x}_t \) is \( r_t \ast h_t \) concatenated with the input, \( x_t \).

3.3.3 Stateful Recurrent Neural Networks

A long input sequence can be split into multiple smaller samples. These samples can then be fed in order through a network, where information is retained between the samples by keeping the internal state and previous output of the recurrent neural network between the samples [4]. This is referred to as stateful training, which allows for multiple gradient updates for each input, rather than one update for the entire input sequence.

3.4 Hierarchical Attention Networks

A document classification task can be approached using Hierarchical Attention Networks [23]. The document is then split into sentences, and each sentence is divided into words. To be applicable to crash reports, the lines in the crash report are treated as equivalent to sentences.

3.4.1 Word Encoder

Each word in the input is embedded as a vector, by applying word embedding [23]. Each sentence is then passed through a bidirectional RNN with a non-linear activation function, GRU [23] or LSTM, and the temporary output at each index is used as the encoding of the word at the same index. Here, the temporary output is a concatenation of the forward RNN and the backward RNN.

3.4.2 Word Attention

To emphasize words that are important while dampening the impact of redundant words, an attention mechanism is applied over all words in a given sentence [23]. The input is then weighed by their corresponding attention weights, in order to form an aggregated sentence.
vector. By keeping the sentence vector at the same length as the word vectors, a sentence vector can essentially be seen as one-word summary of the entire sentence. Specifically, this is calculated according to:

\[
\begin{align*}
    u_{it} &= \tanh(W_w h_{it} + b_w) \\
    \alpha_{it} &= \frac{\exp(u_{it}^T u_w)}{\sum_t \exp(u_{it}^T u_w)} \\
    s_i &= \sum_t \alpha_{it} h_{it}
\end{align*}
\]

Where \( h_{it} \) is the output from the word-encoder described above, for word \( t \) in sentence \( s_i \). \( W_w \) is the weight matrix of the attention mechanism’s internal Multi Layer Perceptron, with \( b_w \) as its respective bias. \( u_{it} \) is the hidden representation of the word, \( u_w \) is an internal context vector which is used to calculate the importance of a word, given its hidden representation. A normalized attention, \( \alpha_{it} \), is then calculated for each word. The aggregated sentence vector, \( s_i \), is then calculated by weighing each word with its word attention. All internal weights are randomly initialized and learned during the training of the network.

### 3.4.3 Sentence Encoder

Each sentence (that is, each line in the crash report) is then encoded by applying a bidirectional RNN, using GRU\[23\] or LSTM, to all sentence vectors, \( s_i \), in the document.

### 3.4.4 Sentence Attention

Similarly, to emphasize sentences which are important whilst dampening the impact of redundant sentences, an attention mechanism is applied over all sentences in a document \[23\]. Each sentence gets encoded and weighed by its sentence based context vector, in order to derive the attention for each sentence. A document based vector is then calculated as an aggregation, where each sentence vector is weighed by its respective attention \[23\]. Formally, this is done by:

\[
\begin{align*}
    u_i &= \tanh(W_s h_i + b_s) \\
    \alpha_i &= \frac{\exp(u_i^T u_s)}{\sum_i \exp(u_i^T u_s)} \\
    d &= \sum_i \alpha_i h_i
\end{align*}
\]

Where \( h_i \) is the output from the sentence-encoder, for sentence \( i \). \( W_s \) is the weight matrix of the sentence based attention mechanism’s internal MLP, with \( b_s \) as its respective bias. \( u_s \) is the sentence based context vector, which essentially answers the question "what is an informative sentence", whilst \( u_i \) is the hidden representation of the sentence. \( \alpha_i \) is the attention assigned to sentence \( i \). Finally, the document vector, \( d \), is calculated as a sum, where each sentence vector is weighed by its respective attention.
3.4.5 Classification

To classify the document, a simple MLP with a softmax function is used, with the document vector, $d$, as input.

$$p = \text{softmax}(W_c d + b_c)$$

Where $W_c$ and $b_c$ are, respectively, the weight matrix and the biases for the final MLP.
Chapter 4
Method

In order to perform an analysis, all crash reports and their corresponding bug reports had to be collected. Redundant data was filtered out from the crash reports. The filtered out crash reports were then separately tokenized on a character-by-character and word-by-word basis. The processed crash reports were then split into a stratified training and validation set, where 30% of the data was used for validation.

An input routine loading batches of crash reports and their respective labels was created, as the data was too large to fit in memory.

A Hierarchical Attention Network [23] was implemented in Keras [7]. The resulting model was then trained on the processed data with categorical cross entropy as loss function, using the Adam optimizer [13].

4.1 Data Handling

The crash reports and the bug reports are stored in different locations. The data therefore had to be obtained, merged and then processed before being used to train a model.

4.1.1 Obtaining the Data

The crash reports were sampled over a 2 year period. All reports were at least one month old. This ensures that each crash report has been appropriately processed, and that the linked bug report is stable.

The bug reports that were linked to the collected crash reports were retrieved from a separate internal database. For each bug report, fields specifying Severity and Responsible Group were saved.
4. Method

4.1.2 Cleaning the Data

Each crash report holds a multitude of different unique tokens. These may for example be hexadecimal values for different registers, or unique numbers for thread id’s. Finding out which of these are important may prove more difficult if a large amount of tokens are redundant. Furthermore the sheer length of each crash report may for some types of models be too long, and require some form of truncation. Hence, this motivates us to devise some form of routine for cleaning and truncating each crash report.

Each filter that was applied to the reports was tested on several crash reports, and the end result was compared to the initial report. If the end result appeared to contain the majority of the information contained in the initial report, whilst reducing the size to a significant degree, the filter was kept.

To begin with, a crude cleanup was used. All digits were removed, including timestamps, unique id’s, and register values. The logs were thereafter truncated to fit the desired length of the current model.

Further on, specific timestamps and their corresponding surrounding characters were removed. Empty lines resulting from cleanup were removed. In the Android-related crash report, a large portion of the messages were sent by the Android Log Info. Typically these log entries hold very little information regarding the issue caused by the bug. Hence, these were removed from the logs.

4.1.3 Creating Training/Validation-split

At first a naive split of the data was used. The data was completely randomly split into two portions. One portion was exclusively used for training the model, and the other for validating the model.

Due to the nature of the dataset, different crash reports may be very closely related, and by practical measures identical. Hence, the results found when doing the naive split were overly optimistic, and did not generalize as well as the results seemed to show. This called for the need of using the crash group specified by the regex parser, in order to determine whether two crash reports actually should be treated as the same or not. The crash groups, and all crash reports contained within each respective group, were therefore split into a training and validation set.

4.1.4 Stratified Training/Validation-split

Naively splitting the data on crash groups into two different datasets may lead to strong discrepancies between the datasets. For example, the proportion of each target value, may be completely skewed between the two training sets. In order to counter this, a stratified training and validation split was used. This yields two datasets, which holds the same fraction of crash groups for each respective target value. Each crash group may contain a different amount of crash reports, but the stratified split ensures a randomized split, which limits the discrepancy between the two datasets.
In practice, the difference between the amount of crash reports, for each target value, within each dataset, was never seen as an issue after relying on the stratified training/validation-split.

4.1.5 Tokenizing the data

After crash reports were cleaned, the information in each crash report was split into tokens. This was done by going through all crash reports and mapping each new token to a new identifying number. A count of occurrences was kept for each token. In the event of a restriction on the vocabulary size, the least frequent tokens were the first to be removed.

The tokenization was done separately on both a character-by-character level, and a word-by-word level. For both of these methods, the tokenization was kept case-insensitive. During the word-by-word tokenization, each word was taken as any token which is separated by one or more filtered out character. As a result, the example "java.lang.NullPointerException() on line"

would be separated into the tokens

"java", "lang", "nullpointerexception", "on", "line".

Subsequently, these tokens would get a unique identifying id. In the event that these tokens would be the only tokens in the crash reports, the resulting id’s would simply be 1, 2, 3, 4, and 5 respectively, as 0 is a token which is reserved for previously unseen tokens.

4.1.6 Parsing out Meta Fields

There are a variety of different types of crashes, and the logs therefore have a quite variable structure. There is however some reoccurring meta information which is logged in a subset of the crash reports. For example, in some of the crash reports the following fields can be found:

- **Process**: `com.android.developer.sample`
- **Flags**: `0xef f1a31`
- **Package**: `com.android.developer v12312 (1.2.3)`
- **Activity**: `com.android.developer.sample.MainActivity`
- **Subject**: `act = android.intent.action.Boot_COMPLETED`

These fields seemed to be promising features to feed to a network in order to increase the predictability of the crash report. As such, all similar types of meta fields that could be found inside any log were gathered and moved into the section that would be used as a basis of the input for the crash reports.

Due to each field being found in a minority of the total crash reports, each field was not represented as a one-hot-encoded input vector, but rather as any other text.
4. Method

4.2 HAN Implementation

The Hierarchical Attention Network suggested by Yang et al.[23] was implemented in Keras [7]. The attention mechanism described in the paper was implemented as a subclass of the Keras Layer. This allows the attention mechanism to be added to any arbitrary Keras model by simply adding an Attention Layer.

4.2.1 Implementation Validation

To validate the correctness of our HAN implementation, the Yelp-2018 [10] dataset was used. This dataset consists of 5,261,671 reviews from yelp.com. The reviews consist of both a description in plain text, as well as a rating from 1-5 stars. Typically, these reviews’ texts consist of multiple sentences, and all reviews are written in English.

A classification of a review’s rating based on its corresponding text is used to validate our HAN implementation. The results from training our HAN implementation are compared with several different configurations of CNN, and LSTM models.

Due to the size of the Yelp-2018 dataset, training one network took between 4 to 12 hours. All training was done on a single GPU.

4.3 Model Selection

For each machine learning model there are typically multiple types of parameters that can be changed. This includes the size of each layer, the amount of layers used, learning rate used during training, optimizer used during training, and activation functions in each respective layer. To find a feasible network topology as well as network parameters, multiple combinations were used to train a model from scratch. Both the results and the history of the training loss and training accuracy were inspected and compared to previous attempts, in order to determine whether any progression had been made.

4.4 Edit Distance

To rule out that crash reports within a crash group are too homogeneous, the edit distance within and in between different crash groups were compared. Due to the length of the crash reports, this was done with a subsample of 80 crash reports, from 4 different crash groups. The first 10,000 characters from each crash report were used for the comparison.

Edit distance is calculated as the smallest total amount of insertions, deletions, and substitutions needed in order to change one string into another. For example, the edit distance between

"com.android.developer.sample"

and

"com.android.developer.main"

is 5. As it at a minimum requires the substitution of three characters, as well as the deletion of two characters in order to reach an identical string.
4.4 Edit Distance

The comparison of this small subsample took roughly 4 days to complete, and was left unattended for this period of time.
Chapter 5

Results & Discussion

The final results on both Severity and Responsible Group classification from crash reports are presented in this chapter. To reach these results, a lot of data exploration was needed. The non-specific findings from the data exploration are presented below. Furthermore, to validate our implementation, another large-scale text based classification problem was used.

5.1 Crash Report Classification

An overview of the obtained results can be seen in table 5.1. The highest accuracy was found using a Hierarchical Attention Network model for both Severity as well as for Responsible Group. The achieved results were 72.5% accuracy on Severity prediction and 51.4% accuracy on Responsible Group prediction, using Hierarchical Attention Network.

5.1.1 Applicability

The results show promising learning and classification of crash reports, without any specific domain knowledge. However, the idea that Severity prediction could be used in order to sort new and incoming crash reports is not deemed to be feasible with the achieved results. The accuracy of the prediction is simply too low in order for one to rely on a system based on its predictions. The Severity prediction is deemed to offer only a rough estimation, and unfortunately it is not particularly good at finding the most (or least) important crash reports. Hence, further improvements would have to be made before such a preliminary ranking would yield any significant value to the process of classifying crash reports.

In the dataset used, there are about 140 different groups available for the Responsible Group prediction. As such, having an accuracy of 51% is by no means trivial, and strongly indicates that the network has managed to learn some tendencies within the data.
### 5. Results & Discussion

<table>
<thead>
<tr>
<th>model</th>
<th>accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNN</td>
<td>69.1</td>
</tr>
<tr>
<td>CNN, char-by-char</td>
<td>68.0</td>
</tr>
<tr>
<td>LSTM, stateful</td>
<td>60.3</td>
</tr>
<tr>
<td>bLSTM, 1 layer</td>
<td>69.2</td>
</tr>
<tr>
<td>bLSTM, multilayer</td>
<td>69.8</td>
</tr>
<tr>
<td>bLSTM, attention</td>
<td>71.0</td>
</tr>
<tr>
<td>bGRU, multilayer</td>
<td>70.1</td>
</tr>
<tr>
<td>bGRU, attention</td>
<td>71.1</td>
</tr>
<tr>
<td>HAN</td>
<td><strong>72.5</strong></td>
</tr>
</tbody>
</table>

**Table 5.1:** The accuracies, in percentage, obtained when training numerous different types of network structures. When unspecified, the input is on a word-by-word basis. For HAN models, the input was fed on a word-by-word basis, and the *attention* was calculated on both a word-by-word and line-by-line basis.

A manual inspection of the logs shows that the task of classifying *Responsible Group* is nontrivial. Hence, an accuracy of 51% without any specific domain knowledge is very promising. It could be that this result offers enough insight in order to be used as a decision support tool. Potentially, offering either a reduction in the time to group similar types of crashes, or an increase to the degree of which similar crashes are grouped.

With this said, it may be that some further improvements would have to be made before it is possible to consistently use our results in order to either save classification time or increase the degree of grouping. Furthermore, it is reasonable to assume that utilizing domain knowledge to shorten the crash reports would increase the amount of relevant information fed to a network, and as such could significantly improve the obtained results.

#### 5.2 HAN Implementation Validation

<table>
<thead>
<tr>
<th>model</th>
<th>accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNN</td>
<td><strong>64.6</strong></td>
</tr>
<tr>
<td>bLSTM, 1 layer</td>
<td>67.0</td>
</tr>
<tr>
<td>bLSTM, multilayer</td>
<td>67.7</td>
</tr>
<tr>
<td>HAN</td>
<td><strong>70.4</strong></td>
</tr>
</tbody>
</table>

**Table 5.2:** Validation accuracy, in percentage, obtained when training a network structure on the *Yelp-2018* dataset.

The results from validating our implementation of the *Hierarchical Attention Network* on the *Yelp-2018* dataset strongly indicates that our implementation yields significant improvements for a text based classification problem. Each of the network structures used for comparison, CNN and bLSTM models, were trained for multiple layers, layer sizes,
and hyper parameters. The best results from these separate attempts are reported in table 5.2. Comparatively, each of the HAN implementation trainings resulted in a validation accuracy above 70%. Hence, our results strongly suggest that our implementation is functioning.

5.2.1 Validation Selection

The Yelp-2018 dataset was chosen as the problem is about using text to classify a rating. This strongly resembles our approach of classifying Severity based on a long text from a crash report. Furthermore, the Yelp-2018 dataset typically consist of reviews which span multiple sentences, making it feasible to run our whole HAN implementation, consisting of both a sentence-by-sentence attention, as well as a word-by-word attention.

In comparison, when classifying crash reports, the sentence-by-sentence attention can instead be done over a line-by-line basis. This, as each line typically contains information which may or may not be linked to the previous or subsequent lines. Furthermore, a line may in many of the cases be composed of a log entry written in English. Typically, this resembles with an arbitrary sentence.

As a comparison, Yang et al.[23] reported a 71% accuracy on the Yelp-2015 dataset. Which is a preceding dataset to the Yelp-2018 dataset which builds on the same principles. The Yelp-2015 dataset does however only contain 1,569,264 reviews. Hence, it ought to be possible to receive a significantly higher accuracy on the nearly four times larger Yelp-2018 dataset. Although we do not achieve a higher accuracy than that which was reported by Yang et al. on the Yelp-2015, we do not do any other text based optimizations which ought to further increase the results on the Yelp-2018 ranking classification. Hence we do not do an embedding initialization based on Word2Vec [16], nor stemming of English words. As such, we find our results comparable and likely to be representative for the methods chosen.

5.3 Correlation Between Crash Reports

Early attempts of training a model with a bLSTM, bGRU, and CNN networks indicated some issues of overfitting when all crash reports were used. It appeared as if lowering the amount of allowed crash reports per crash group would give better results. This is unexpected, as a larger dataset tend to give better results. To our understanding, crash reports should never be exactly the same but rather have quite significant discrepancies. Normally such discrepancies tend to help the network in its training, as it forces the network to more properly learn how to generalize.

In order to get some understanding of the similarities that exists between different crash reports, around 100 different crash reports were manually inspected. The majority of these crash reports were from the same crash groups.

A comparison seemed to indicate that the majority of all crash reports, even within a crash group, have a high variation. However, some near exact duplicates, that seemed only to differ by something as small as a newline character, were also found. Overall it seemed as if the differences between crash reports were significant enough in order to preclude any form of overlapping input data, both within and in between different crash groups.
5.3.1 Edit Distance

Since the preliminary comparison seems to indicate that our model is the issue behind the overfitting, and not the correlation within the input data, we continued our investigation into the degree of similarity amongst crash reports. This, as we would much prefer to use all available data, as long as it contains sufficient variation.

As such, the edit distance within a subsample of crash reports was compared. The results of this comparison can be seen in figure 5.1.

![Figure 5.1: The edit distance between different crash reports, visualized within and in between different crash groups. Normalized by the length of the crash report.](image)

In figure 5.1 we can see that the crash reports among different crash groups tend to have a very high dissimilarity. This is expected, as they are completely different types of crashes, caused by completely different types of bugs.

We can also see that there are some crash reports within a crash group that are next to identical. In total these add up to about 5% of the compared examples. However, the majority of the crash reports, even within the same crash group, tend to have quite significant dissimilarities. At the same time, crash reports within a crash group are being highly correlated, compared to crash reports in-between different crash groups.

The higher similarity within a crash group, and the limited amount of nearly identical crash reports, strengthen our hypothesis that it is feasible to use all of the crash reports available.
Although it would be ideal to not have any identical, or next to identical, crash reports, we deem it highly improbable that a 5% overlap would contribute to an amount of overfitting that is unfeasible. The potential benefit of sorting out these near identical crash reports would hence be too low for the effort that it would consume. As such, the crash reports were used in their entirety when training subsequent models. Since the previous models had been overfitting for this setup, this encouraged us to find a structurally better machine learning technique for classifying the crash reports.

5.4 Training

Attempts using bLSTM, bGRU, and CNN networks seemed to give roughly the same accuracy for a lot of different attempts, where amount of layers, size of layers, and hyperparameters were tuned. Typically, a validation accuracy of 68% ± 1% would be achieved. A maximum of 70% validation accuracy was found. However, the results tend to be unstable, and fluctuate between attempts, as well as between epochs. Since the loss for the higher accuracies were not found to be stable, we deem the whole results as unstable, and merely coincidentally better or worse.

Lowering the learning rate by a factor 50 gave much more stable results. However, this would consistently only yield a 68% validation accuracy. Furthermore, the history from these trainings would show that the only increase in accuracy is over one epoch. Suggesting that the network only learns one trick. After that, the validation loss consistently increases, while the validation accuracy remains stable. This can be seen in figure 5.2.

Interestingly enough, using the HAN model resulted in a validation accuracy which during training was comparatively much more stable, both between attempts, and between epochs. This, as the volatile tendencies of having a validation accuracy and validation loss which increased and decreased by several percentage points between epochs, was no longer present to the same extent. Indicating a more stable training, and consistent results, compared to the ones achieved by the bLSTM, bGRU, and CNN networks.

5.4.1 Training/Validation-split

Early attempts were made by randomly selecting a subset of the crash reports to be used for validation. Hence, a crash group may be represented in both the training- and validation-set. Although this did not seem to be crucial at first, early attempts using this method gave a validation accuracy of over 90% on Severity prediction. Had this been reliably applicable on new crash reports, it would have been a stellar result. However, the similarities that exists between crash reports within the same crash group leads to some highly similar crash reports existing in both the validation and training data. Hence, this result can not be trusted, nor used.

To counter the problem where similar crash reports exists both in the training and validation set, a stratified training/validation-split was used. This ensures not only that crash reports that exist in the validation set, are absent from the training set, but also that the underlying issue which causes the crash report is most likely absent from the training-set as well. Hence, this leads to a significantly harder problem, yet a much more realistic and applicable one.
5. Results & Discussion

Figure 5.2: The accuracy and loss measured during training. The only increase in terms of validation accuracy is achieved during one epoch. Suggesting that a simple trick is learnt, and not much more than so.
5.4.2 Stateful RNN

The idea of stateful RNN is to split up a continuous input into multiple training examples with the same label. This was attempted as a way of handling the text composing a crash report. As reported in table 5.1, this was not very successful. The reason to this is believed to be that only a fraction of a crash report is actually used for its final classification. As such, a stateful RNN will train on a large amount of examples, which are by no means connected to its label. Hence, the idea of proceeding with this form of network structure was dropped.

5.5 Meta Fields

All meta fields that were available in each log, such as Process:, were included in the input which was fed to the network. It may seem appropriate to one-hot encode each of these fields, so that the network will have an easier time interpreting the information that it is given. However, since there was no field which occurred in the majority of the crash reports, this would lead to a substantial amount of training on crash reports for which the features simply were completely unknown. Furthermore, it would also allow the network to take a shortcut, and simply predict the output as a probability given the values of these meta fields. This may hinder the network from learning anything other than that which can be understood from these meta fields.

Although it may seem feasible that meta fields in general can be used in order to yield a somewhat accurate prediction of the Severity or Responsible Group for a crash report, this is a method whose maximum potential is highly limited. By no means does the statistical Severity of a crash report having certain of these fields truly indicate whether a new and unseen crash report is actually of that Severity. Hence, although it may be compelling to focus on easily obtainable information which may say quite a lot about the crashes, this information is neither widely enough available, nor precise enough, to yield a sufficiently good result.

It is quite feasible for a human to scroll through a crash report and quickly grasp what information these fields hold, and as such, giving a summary of these, or simply making a prediction based on this, was deemed to have little value. Therefore, the meta fields were included in the input fed to the network, but the fields were not treated as a specific one-hot encoded, categorical input vector.

5.6 Crash Group Sizes

An inspection of the available crash reports seem to indicate that there are some correlation between how many times a certain type of crash appears, i.e. the amount of crash reports in a crash group, and the Severity of the crash. Figure 5.3 visualizes the number of crash reports per crash group, for each of the available Severity values.

There is a clear tendency that a highly severe crash causes more crashes. Interestingly enough, the least severe crashes tend to also be overrepresented when it comes to a really high amount of crashes for its average crash group. Indicating that a crash which is found many times, is typically highly severe, or not critical at all.
5. Results & Discussion

5.7 Crash Types

The data collected consists of various types of crash reports. Some crash reports are generated from crashes where the underlying issue was in hardware whilst others are from JAVA or kernel crashes. This tends to generate different forms of crash reports. A certain type of crash tends to result in crashes that somewhat follow the same format. However, this format is changing both for the same type of crash, as well as for different types of crashes.

It would be ideal to treat each separate type of crash with a separately trained network. However, as we had to create a stratified training- and validation-split in order to perform proper training, using a separate network for each type of crash would presumably result in too little input data. As such, we opted for the alternative where all different types of crashes are used in order to train one larger network.

As there is a rather heavy variation both within the types of crashes, as well as in between, we also saw the potential benefit of additional learning from similarly formatted crash reports of a different crash type. Furthermore, our approach generalizes a bit better,
as it requires less human interaction and therefore less domain knowledge, in order to make a prediction of a new, arbitrary crash report.

The potential benefit of training different types of networks, for each respective type of crash, was not investigated due to the limitations of the dataset sizes, as well as the major variations within each crash type.

## 5.8 Future Works

Although this method is not deemed to be sufficient in order to produce a stand-alone tool to perform autonomous categorization of software defects, it is done without any specific domain knowledge. This implies that more extensive and diligent feature extraction, based on domain knowledge, could very well significantly improve the predictability. Potentially leading to a decision support tool which could be used to give reliable aid in the process of categorizing software defects.

Furthermore, the data which has been used has been formatted in a way which targets a human user. As such it may very well be possible to devise a format which is more suitable for a machine learning approach.

Rather than finding similar crash reports through looking at which personnel ended up resolving the issue, it may be more feasible to look at crash reports which seem to have a similar underlying problem. Using this information could allow for a more specific classification problem, resulting in a much larger output space.

However, classifying crash reports which have historically proven to stem from the same type of underlying issue is not enough in order to obtain any form of applicability of the system, as the main idea of the system would be to provide information for new, previously unseen crash reports. Ideally, with an underlying issue that has never been seen before. Hence, the issue at hand would become something that more resembles a clustering problem. It may here be feasible to use a network trained on labeled historical data, in order to be able to encode the information from a crash report. However, when new crash reports are classified or clustered, small variations would to a larger extent affect the test accuracy of the trained models. Hence, extensive human verification of such a system would be essential. For this reason, it was not feasible to attempt such an approach for this report.
Chapter 6
Conclusions

We compared different types of machine learning techniques in order to classify automatically generated crash reports with very little domain knowledge. The results strongly indicate that a significant amount of information can be learnt from the crash reports using our proposed methods. Furthermore, our result indicate that it could be feasible to use a similar type of approach in order to create a decision support tool, which can be used to both reduce the time it takes to categorize crash reports as well as increasing the degree to which such a grouping is made.

The crash reports used vary in both structure and information. As such, we attempted a very general way of classifying crash reports using a long, text-based approach. To simulate a more realistic scenario, all information which could be used in order to group together crash reports stemming from the same underlying issue was used when dividing the dataset into a training and validation set.

Ultimately, to further improve the results beyond what CNN and RNN structures can seemingly perform, significant improvements to the data handling are deemed necessary. In short, we believe that a more general and consistent way of reducing the information in a crash report is needed in order to achieve results good enough for a deployable tool. We propose that this could be done by utilizing the domain knowledge of the people who currently work with categorizing software defects through incoming crash reports.
Bibliography


Kategorisering av Mjukvarufel via Maskininlärning

Vid storskalig utveckling av mjukvara är det oundvikligt att introducera buggar av olika slag. I det här arbetet har det undersökts hur maskininlärning kan användas för att underlätta processen av att åtgärda mjukvarufel.

När en krasch sker på en av Sony Mobile’s testtelefoner, rapporteras automatiskt en kraschrapport till en server. Dessa kraschrapporter används som underlag när testare skriver ihop en buggrapport som beskrivs vad det är för typ av fel, hur allvarligt felet är, och vilken grupp av utvecklare som kan tänkas åtgärda problemet.

Då en bugg tenderar att leda till flera krascher finns det ett stort intresse i att gruppera liknande typer av kraschrapporter för att minska tiden det tar att hantera alla inkommande kraschrapporter. Detta kan låta enkelt, men kraschrapporterna innehåller tusentals loggade händelser i godtycklig ordning. Därmed är all form av automation som kan underlätta det här arbetet av intresse.

Genom att använda oss av de buggrapporter som skapats har vi undersökt hur en övervakad maskininlärning kan användas för att hitta samband i kraschrapporterna, vilka i sin tur kan användas som ett underlag till de som i dagsläget manuellt klassificerar krascher på Sony Mobile.

Då kraschrapporterna varierar i väldigt hög grad, både i struktur och information, hanterade vi dem som långa texter. Dessa texter klassificeras sedan baserat på vad som beskrivs i dess respektive buggrapport. Man tittar både på Severity och Responsible Group, vilket motsvarar hur allvarlig en krasch är på en skala 1-4, respektive vem som ansvarar för att åtgärda problemet.

Huvudsakligen användes Recurrent Neural Networks, vilka har en form av återkoppling och därigenom ett slags ’minne’ för att bättre kunna göra en klassificering baserat på det som nyligen setts i exempelvis en text.

For att ytterligare förbättra klassificeringen implementerades ett Hierarchical Attention Network. Vilket i grunden är ett Recurrent Neural Network med en form av extrainsatt uppmärksamhetsmekanism, vilken först tar fram de ord som är intressanta, och därefter gör en klassificering baserat på detta.

När en ny, godtycklig, och aldrig tidigare sedd kraschrapport och kraschorsak inkommer kan vi med 72% träffsäkerhet förutsäga hur allvarlig kraschen är, och med 51% träffsäkerhet förutsäga vilken grupp som kommer att bära ansvar för att hantera problemet.

Detta kan användas som ett beslutsunderlag för att lättare se vilka inkommande kraschrapporter som har mer gemensamt med varandra, och därigenom antingen förkorta tiden för den manuella hanteringen, eller förbättra tillförlitligheten. Däremot finns det fortfarande begränsningar då vårt resultat ej är tillräckligt bra för att konsekvent, fullt ut, lita på det. Därmed finns det behov av förbättringar, och vi tror att detta är möjligt då vi i vårt arbete använt oss av väldigt begränsad domänkunskap. Genom att använda sig av expertkunskap för att ta fram de absolut mest relevanta raderna i en kraschrapport, och därefter använda dessa som indata för maskininlärning, tror vi att resultatet kan bli tillräckligt bra för att skapa ett tillförlitligt system som underlätta i processen av att åtgärda mjukvarufel.