Delft University of Technology, Bachelor Seminar of Computer Science and Engineering

# Automated Deep Learning based on syntactic context Features for Log Level Recommendation

Erwin van Dam

June 26, 2021

**Responsible Professor:** Dr. Maurício Finavaro Aniche

**Supervisor:** Jeanderson Barros Cândido

**Peer group members:** Kostas Lyrakis, Liudas Mikalauskas

**Abstract**

Automated log level recommendation is a growing area of research in the field of logging. Logs are essential in software maintenance. Log levels influence the severity of the logs being printed. Recent studies have investigated different metrics for automated log level recommendation. Recently, a paper was published using automated deep learning based on syntactic context features for log level recommendation. The paper shows promising results, both for within-system evaluations and cross-system evaluations. Here, the results posed by that paper are validated by reconstructing the model from the paper. Furthermore, the model performance is evaluated on different features, for instance, the containing block type. This study demonstrates that automated deep learning based on syntactic context features for log level recommendation certainly provides promising results. The outcomes even indicate that cross-system performance resembles within-system performance. However, this paper also indicates that the model cannot predict log levels for unseen systems. In conclusion, this paper validates that the current methodologies show potential for future research, but that the model is not ready for production. More research is necessary to transform the current algorithm into a production ready version of the algorithm.

## 1 Introduction

Logs are essential in software maintenance. Log statements record valuable run-time information [19]. Several common motives behind logging are testing, failure diagnosis and program comprehension [20].

*Log levels* influence the severity of the logs being printed. Using log levels, users can decide to print the more severe log levels while suppressing the more verbose log levels [19]. The possible log levels are *trace, debug, info, warn, error* and *fatal*. These log levels are based on the logging library used throughout the projects studied in this paper: Apache Log4j [10]. Trace is the most verbose but the least severe log level and fatal is the least verbose but the most severe log level. These log levels are ordinal in nature. When log level warn is chosen, error and fatal are printed too as these log levels are more severe than warn [19].

The decision upon log level is essential. If the chosen log level is too high, the overhead of log management and log analysis increases. On the other hand, if the chosen log level is too low this results in missing information regarding run-time failures [20].

Automated log level recommendation would help developers make informed decisions on the proper log level for a logging statement. It is challenging for developers to choose the proper log level [20]. Currently, developers mostly rely on their experience to determine what log level to use [4]. Automated log level recommendation would not only alleviate the insertion of a new logging statement, but could also assist in the case of an alteration of log level for an existing log statement [19]. This way, developers can shift their focus from deciding upon the proper log level to other tasks. Furthermore, automated log level recommendation would result in common logging practises throughout entire projects, which is very important [19]. Thus, both developers, project owners and users could benefit from automated log recommendation.

There are several different research areas in the field of logging [5]. This paper focuses on the area of logging practises. More specifically, this paper is a replication of the paper by Li et al. [20]. Research upon logging practises is the youngest area of study in the field of logging [5]. This area is quickly expanding. Recently, several studies have investigated potential algorithms for automated log level recommendation. Below, two of these studies are further elaborated.

Li et al. [19] developed one of the first automated log level recommendation systems. In their own words: "To the best of our knowledge, there exists no prior research regarding log level guidelines". The paper starts with an investigation of the distribution of log levels throughout several projects. Their findings show that log levels are distributed differently depending on the containing block type of a log statement. The statistics show that the distribution of log levels is different in for instance catch blocks compared to for-loops. As a result, the paper concludes that containing block type, in combination with the type of exception caught for catch blocks, is a good metric for log level recommendation. This metric is compared to numerous primitive metrics, which can be divided into *Logging statement metrics, Containing block metrics, File metrics, Change metrics and Historical metrics*. The paper combines the best metrics into an Ordinal Regression model.
Li et al. [19] certainly developed one of the first stable automated log level recommendation systems, thereby forming the foundation of this area of study. However, there are certain drawbacks in the approach of the paper. First of all, the baselines used are a Random Guessing Model and a Naive Model based on the proportional distribution of log levels. Beating these baselines does not necessarily mean that the model is optimal. Do note that there was not a better baseline to be used at that time. Furthermore, the approach in the paper with the large amount of metrics is very project dependent. This resulted in a "significant performance degradation" once the model was evaluated on another project than the project the model was trained on.

More recently, Li et al. [20] came up with a new model for automated log level recommendation. The paper proposes an ordinal approach with two complex metrics (*syntactic context* and *log message features*) using a recurrent neural network (*RNN*). In their findings, the syntactic context is a key factor in log level recommendation. The paper states that

their model using only the syntactic context outperforms the Ordinal Regression model proposed by Li et al. [19] for any dataset the models were evaluated against. Furthermore, the ordinal encoded RNN always outperforms a one-hot encoded RNN. Even more, cross-system performance produced promising results.
The problem with this paper is that most of the results are not properly validated. As the paper mentions, the hyper-parameters should be optimised for even better results. Furthermore, the paper emphasizes that the containing block type is a very important metric for log level recommendation. Although the model implicitly captures this metric, the paper does not elaborate on whether it is better to capture this metric implicitly, or to precompute the containing block type and pass it to the model as an input parameter explicitly.

In this paper, the results described in the paper by Li et al. [20] are validated. This paper concludes that automated deep learning based on syntactic context features for log level recommendation certainly is a promising option for automated log level recommendation. This paper encourages for more research on this methodology, to validate whether this method would work with other systems, languages, or contexts. This paper supposes that the algorithm proposed in the paper by Li et al. [20] forms an important step in this direction, but that more research is needed to transform the current algorithm into a production ready version of the algorithm. To validate the paper by Li et al. [20], this paper answers two research questions:

- RQ1: What is the performance of automated deep learning based on syntactic context features for log level recommendation?
- RQ2: Is the approach able to perform cross-system recommendations?

This paper is structured as follows. Section 2 will work out the main concepts of automated deep learning based on syntactic context features for log level recommendation. After section 2, section 3 describes the model created for this paper. Here the different intermediate steps from the algorithm are outlined. Next, section 4 reports the results of the model proposed in this paper. These results include the cross-system results regarding the second research question as introduced in section 1. Then, section 5 reflects on the ethical sides of this research, for instance the reproducibility of the research. Afterwards, section 6 discusses the results and the design decisions leading to the results. Finally, section 7 comes back with the research questions posed in the introduction and finishes off with the corresponding conclusions. This conclusion also covers the most interesting findings, main problems, and future recommendations with regard to the research done in this paper.

# 2 Concepts behind Log Level Recommendation

The model proposed in this paper is based upon several key concepts. In this section, the most important concepts are explained in more detail. Those concepts are: *deep learning*, *Ordinality* and the syntactic context as mentioned in section 1.

## 2.1 Deep Learning

Deep learning is a specific group of algorithms in the class of Machine Learning algorithms. Machine Learning algorithms usually have “the ability to learn and enhance from experience automatically without being specifically programmed” [27]. Deep learning provides an artificial *neural network* which the user can build up using layers [27]. The concept behind neural

Table 1: Ordinally encoded log levels compared to one-hot encoded log levels

| | *ordinally encoded* | *one − hot encoded* |
|---|---|---|
| *trace* | [1, 0, 0, 0, 0] | [1, 0, 0, 0, 0] |
| *debug* | [1, 1, 0, 0, 0] | [0, 1, 0, 0, 0] |
| *info* | [1, 1, 1, 0, 0] | [0, 0, 1, 0, 0] |
| *warn* | [1, 1, 1, 1, 0] | [0, 0, 0, 1, 0] |
| *error* | [1, 1, 1, 1, 1] | [0, 0, 0, 0, 1] |

networks mimics the way the human brain works. A standard neural network can learn the parameters for a mapping function from an input vector to the output vector. The neural network tries to capture the underlying relationships between elements in the input vectors through parameter optimization. Deep learning performs best on large datasets [27]. The quality of the dataset is crucial for the performance of the neural network. The deep learning algorithm used in this paper is supervised. This means that the algorithms' goal is "to learn a function that maps an input to an output based on sample input-output pairs" [27].

The specific algorithm used in this paper is a Bidirectional Long Short-Term Memory (*Bi-LSTM*). The main difference between LSTM and other deep learning algorithms is that LSTM provides feedback links [27]. These feedback links capture the order of sequential data. A normal LSTM can only capture the order of sequential data in one direction. A Bi-LSTM can capture the order of sequential data in both directions. Bi-LSTM's are good at capturing the context of an element in an input vector.

Both the notion that deep learning performs best on large datasets and the notion that a Bi-LSTM can capture the context of an element in an input vector are important notions regarding log level recommendation. Both notions are applied to log level recommendation in future sections. Section 6.2 describes why the notion of a large dataset makes deep learning a good choice for log level recommendation. Section 2.3 explains why the context and its ordering are important for log level recommendation. Both notions combined make Bi-LSTM a good option for log level recommendation.

## 2.2 Ordinality

Log levels are ordinal in nature [19] [20]. This ordinal nature embodies the severity and the verbosity of the log levels. There are multiple log levels. Multinomial classification would be able to perform log level recommendation, but does not capture the ordinal nature of the log levels. Ordinality can be captured by ordinally encoding the log levels instead of one-hot encoding the log levels [20]. Table 1 visualizes both log level encodings. In an ordinally encoded vector, each dimension represents the probability for a certain log level, or a more severe log level. Li et al. [20] compare the model using ordinal encoding with an equal model using one-hot encoding. In their findings, the model using ordinal encoding performs better for any metric on any dataset.

## 2.3 syntactic context

The metric used in this paper is called the syntactic context. Li et al. mentions that "the containing block metrics (...) play the most important role in the ordinal regression models for log levels" [19]. Li et al. state that "the information of logging locations (...) may be related to the decision of log levels" [20]. Both papers independently find that the context of a log statement is an indicator for log level recommendation. The syntactic context has two characteristics: it is **syntactic** in nature and represents the **context** of a log statement. Both characteristics are further elaborated in the paragraphs down below. These characteristics are largely based on the syntactic context as described by Li et al. [20].

**Syntactic**. The input of the model is based on syntactic information, specifically the type of Abstract Syntax Tree (*AST*) nodes in the context. This syntactic approach implicitly filters out most non-structural information. Variable names, for example, `boolean success = false`, form very project-specific information. The decision for variable names largely depends on design choices made by the development team. To abstract away from this project-specific information, the model in this paper uses structural information as input, for example, *VarDeclar* and *BooleanLiteral* for `boolean success = false`. To keep the input clean of non-structural information, certain node types are filtered out, for example *SimpleName* is commonly filtered out, which represents an identifier name.

**Context**. The input of the model is an abstract representation of the control flow from the beginning of the method up till the end of the basic block around a log statement. For this reason, the order of the statements in the context is very important. Log-related information is not part of the control flow of the code and should thus not be in the context. Log-related information in the context could give away the log level. This could result in a model that would be dependent on log-related information in the context. Such a model would under-perform for projects with no log-related information in the context. In order to create a context that only captures information related to the control flow of the code, all log-related information should be removed. For this paper this means that all log levels should be masked and that all log guards should be removed.

The **syntactic** characteristic and the **context** characteristic combined form the syntactic context. This syntactic context is represented by the AST node types in the control flow of the code. This representation does not contain any project-specific information nor log-related information anymore and is thus purely structural in nature.

syntactic context example

```
1  "syntactic_context": [
2      "MethodDeclaration",
3      "TryStmt",
4      "VariableDeclarationExpr",
5      "IfStmt",
6      "LogStmt",
7      "ReturnStmt",
8      "NullLiteralExpr"
9  ]
```

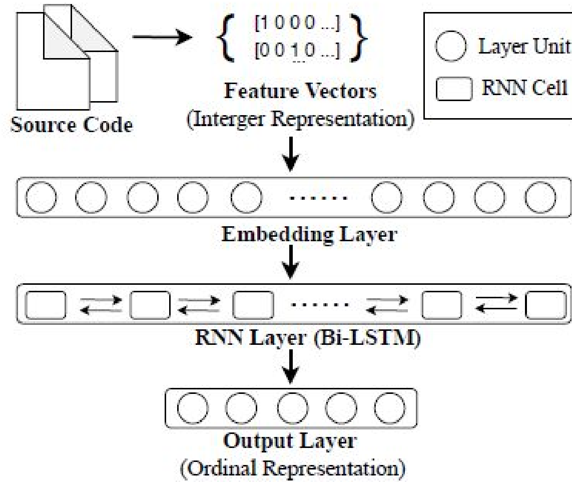


Figure 1: Steps of the algorithm (Li et al. 2021)

# 3 Automated Deep Learning based on syntactic context Features

The algorithm in this paper consists of the following steps (these steps are visualised in Figure 1):

- Dataset Creation
- an Embedding Layer
- a RNN Layer
- an Output Layer

Beneath, the implementation of these steps of the algorithm are described in more detail.

## 3.1 Dataset Creation

The dataset for the model is created from the source code of Apache CloudStack [11] (v4.15.0.0). For the first step of the algorithm, the supervisor of this paper has set up a tool that automatically downloads the source code of Apache CloudStack. Furthermore, this tool extracts all file paths with their corresponding file types from the source code. The possible file types are *PRODUCTION, TEST, DOCUMENTATION* and *BUILD*. This project only uses the PRODUCTION files. In Apache CloudStack, there are no log statements in the DOCUMENTATION and BUILD files. TEST files are not used because of the assumption that developers use different logging standards for code related to testing than for production code.

The code that creates the dataset is reusable for any logging dataset. For this particular project, the syntactic context is the only metric being used. The syntactic context depends on the method around a log statement. However, for example file metrics [19] depend on

Table 2: Number of files, methods, and log statements per file type in Apache CloudStack

| | *files* | *logged files* | *methods* | *logged methods* | *log statements* |
|---|---|---|---|---|---|
| *PRODUCTION* | 5212 | 1114 | 51368 | 5176 | 12825 |
| *TEST* | 554 | 45 | 5074 | 162 | 694 |
| *TOTAL* | 5766 | 1159 | 56442 | 5338 | 13519 |

more than just the method around a log statement. To support these type of metrics, the program creates a JSON [23] array of files. Each file can contain zero or more project-specific file metrics. Furthermore, each file in the JSON array contains an array of methods. Each method can have zero or more project-specific method metrics and has a list of log statements. Each log statement can again contain zero or more project-specific log statement metrics. For traceability, the file path per file, method name per method, and line number per logging statement are always stored. Furthermore, for each file or method the program stores whether the file or method is logged. Lastly, per log statement, the log level is always stored.

Output JSON

```
1  [{
2      "file_path": "download\\cloudstack\\example.java",
3      "is_logged": true,
4      "eventual_file_metric_1": "file_metric_1_value",
5      "methods": [
6          {
7              "method_name": "method_1",
8              "is_logged": true,
9              "eventual_method_metric_1": "method_metric_1_value",
10             "log_statements": [
11                 {
12                     "line_number": 10,
13                     "log_level": "debug",
14                     "eventual_log_metric_1": "log_metric_1_value",
15                 }
16             ]
17         },
18         {
19             "method_name": "method_2",
20             "is_logged": false,
21             "log_statements": []
22         },
23     ]
24 }]
```

The non-metric related code (e.g., reading and parsing files) is separated from the metric

related code (e.g., syntactic context extraction). The program consists of an abstract class that contains all non-metric related code and an implementation of this abstract class containing all metric-related code. In the abstract class, JavaParser [18] creates an AST for each file in the source code. For each AST, the abstract class visits 3 abstract methods: *visitFile, visitMethod* and *visitLogStatement*. These abstract methods can be implemented to create a custom logging dataset. Furthermore, the abstract class stores the fields for traceability as mentioned above. The resulting JSON [23] array of files is stored to file. All methods containing *UnparsableStmt* nodes are removed from the dataset. Above that, all inner methods are kept out of the dataset. This prevents duplicate data in the dataset.

Abstract class Implementation

```
1  @Override
2  JSONObject visitFile(JSONObject fileJSON, CompilationUnit file)
3
4  @Override
5  JSONObject visitMethod(JSONObject methodJSON,
6                         MethodDeclaration method)
7
8  @Override
9  JSONObject visitLogStatement(JSONObject logStatementJSON,
10                        ExpressionStmt logStatement)
```

The syntactic context extraction is based on a paper describing an AST representation for program analysis using Machine Learning [29]. The algorithm uses depth-first traversal over the AST to create the syntactic context. This syntactic context consists of *basic blocks* and other nodes. The basic blocks represent the different *bodies* in code like a *try statement* or a *method declaration*. For these basic block nodes, the header is seperated from the body (e.g., for an if statement, the condition is separated from the body). Furthermore, if there are several branches, only the branch that is an ancestor of the log statement is included in the syntactic context (e.g., for a switch statement, only the switch entry in which the log statement is, goes into the context). The basic blocks included in the algorithm are: *'BlockStmt'*, *MethodDeclaration*, *LambdaExpr*, *TryStmt*, *CatchClause*, *SynchronizedStmt*, *ForStmt*, *ForEachStmt*, *WhileStmt*, *DoStmt*, *IfStmt*, and *SwitchStmt*. The other nodes are included in the syntactic context if the basic block around the node is part of the syntactic context.

JavaParser [18] does not contain a log statement node. For this reason, log statements would end up in the syntactic context as *MethodCallEpxr* nodes. To prevent this, the program keeps track of all log statements in the context. These log statements are later replaced with the label *LogStmt*.

The source paper does not reveal a lot of implementation details about the retrieval of the syntactic context [20]. The paper mentions that AST nodes that do not contain structural information are filtered. Furthermore, log guards are removed. Lastly, the paper describes that all nodes from the beginning of the method up till the end of the basic block around the log statement form the syntactic context. These three features are built into the algorithm of this paper. For the first feature, the user can specify a set of non-structural nodes in the configuration file. These nodes are filtered from the syntactic context. Log

guards are filtered from the context using a regex expression.

Log guard regex

```
String regex = "is(Info|Warn|Debug|Error|Trace)Enabled";
```

By manual investigation, the decision on how to erase all traces of a found log guard from the context depends on the parent node of the found log guard. There are three different scenarios in the source code of Apache CloudStack. First of all, some log guards appear in a variable declaration. The entire variable declaration is removed from the context as it contains log-related information.

Log guard variable declaration example

```
boolean isDebugEnabled = s_logger.isDebugEnabled() // removed
```

The largest group of log guards is directly encapsulated in an if statement. This if statement is removed from the syntactic context, but the body of the if statement is not removed from the syntactic context. If there were no log guards, the if statement would simply be replaced by its body.

Log guard if statement example

```
if (logger.isInfoEnabled) { // => ...body
    ...body
}
```

The last group of log guards is encapsulated in a binary expression. Each of these binary expressions is replaced with the left or right child expression that does not contain the log guard.

Log guard binary expression example

```
if (rows == 0 && s_logger.isDebugEnabled()) { // => if (rows == 0) {
    ...
}
```

The last feature (all nodes from the beginning of the method up till the end of the basic block around a log statement form the syntactic context) is implemented using the basic blocks. If a basic block is not an ancestor of the log statement, then the basic block's header and body do not appear in the syntactic context. If a basic block is a direct ancestor, then all statements in the body of the basic block are put into the syntactic context. Alternatively, if the basic block is an ancestor, but not a direct ancestor, than all statements until the child node that is an ancestor of the log statement are included in the syntactic context.

In order to get more insight into the distribution of the data, two more features are stored per log statement: the containing block type and the *syntactic context size group*. As mentioned in subsection 2.3, both papers independently find that the context of a log statement is an indicator for log level recommendation [20] [19]. The syntactic context size group originates from the midterm presentations. During this meeting, the responsible professor recommended to investigate how the performance of the algorithm changes for different sizes in the dataset. The different groups regarding the size of the context and the different containing block types can be set in the properties file. In section 4, both features are used to get a more detailed overview of how the model performs per containing block type and per syntactic context size group.

Distribution features

```
1  "containing_block_type": "Catch",
2  "syntactic_context_size": "20-30",
```

The last step during dataset creation is to create the training set, validation set, and testing set. For some datasets these sets can be created by randomly dividing the dataset into three groups. For this dataset, the distribution of log levels is important. Random sampling might result in a biased dataset. The log levels would not be distributed equal to the overall dataset distribution, which would not be representative for the original dataset [24]. Therefore, the different datasets are not just created by random sampling, but by sampling the subgroups in the dataset. In other words, each individual dataset is created by random sampling per log level, proportional to the distribution of the entire dataset. This process is called *Stratified Sampling* [8].

The training set, validation set, and testing set are necessary to get an unbiased estimate of the performance of the model [26]. First of all, the training set is used to train the model. The two other datasets are used as a metric for model performance. The idea behind this is that the model cannot be properly validated using the training set, as the model has seen this data before. Using data the model has never seen before to validate or test the performance gives a fair representation of what the model would perform like in reality, as long as the dataset is representative for reality. The distinction between the validation set and the testing set lays in the development stage for which the dataset is used. The validation set is used during training. This dataset is used to validate a round of training. The testing set is used when the entire training process is done.

Stratified sampling is implemented using functions defined in pandas [28]. The implementation results in datasets with an equal distribution. The implementation does not result in disjunctive datasets. Because of the random implementation of pandas' function `dataset.sample()`, an entry can appear in all three datasets. Due to time constraints, this process has not been updated to create disjunctive datasets.

## 3.2 Embedding Layer

The dataset contains vectors consisting of words like *TryStmt*. This data cannot be used as input for the model yet. In order to convert the vectors to input vectors for the model, each vector is translated to a frequency vector. Each dimension in the frequency vector

Table 3: Number of log statements per dataset per log level in Apache CloudStack

| | *trace* | *debug* | *info* | *warn* | *error* |
|---|---|---|---|---|---|
| *Training* | 3241 | 1484 | 1072 | 385 | 1514 |
| *Validation (and Testing)* | 1080 | 495 | 357 | 128 | 505 |
| *All datasets combined* | 5401 | 2474 | 1786 | 641 | 2523 |

corresponds to a specific type of node in the syntactic context [20]. These frequency vectors are most of the times very sparse [25]. There might be several relations between features of these sparse vectors. These relations can scale down the length of the vector [25]. An Embedding layer does just that. As the keras documentation says, an embedding layer "turns positive integers (indexes) into dense vectors of fixed size" [6]. In this paper, the embedding layer turns the positive frequencies of the nodes in the syntactic context into dense vectors of fixed size.

An Embedding layer can be implemented apart from the model or as the first layer in the model. An example of embedding separated from the model is word2vec [22]. This embeds plain English into vectors. There is however no standard embedding for AST nodes. The embedding could still be done outside the model. The embedding would then focus purely on the strongest relations in the vectors. However, by including the embedding layer into the model, keras [6] is able to optimize these embeddings to get better results [9].

The Embedding layer has been implemented using one of keras' [6] layers. This Embedding layer is the first layer in the Sequential model. This layer contains one hyper-parameter that is not input-dependent: the output dimension. In the current version of the model, this parameter is set to 64. This parameter should be optimised in future research.

## 3.3 RNN Layer

The next layer in the algorithm is the RNN layer, more particularly, the Bi-LSTM layer. As described in subsection 2.3, the context and the order of the context are essential to represent the control flow of the code. In the model of this paper, this layer has been implemented by using one of keras' [6] layers. The hyper-parameters (128 as dimension of the hidden states and 0.2 as dropout rate) were taken from the source paper [20].

## 3.4 Output Layer

After the Embedding layer and the Bi-LSTM layer, the output layer is the last layer of the model. This layer converts the output vector of the Bi-LSTM to a vector that represents a log level. As described in subsection 2.2, this output vector is the ordinal representation of a log level. The output layer is a dense layer consisting of a certain amount of units. This amount of units is equal to the amount of log levels included in the configuration file. Each unit in the dense layer represents the probability for one log level. The activation function used in this layer is *sigmoid*, just like the source paper. This function results in a number

between 0 and 1. If every dimension in this vector is thresholded by 0.5, the resulting vector looks like one of the vectors in Table 1. By ordinally decoding this vector of probabilities, the predicted log level is retrieved [20].

### 3.5 Model

The overall model consists mainly of the layers described above. However, the model includes some more hyper-parameters. The loss function is *binary crossentropy*. This hyper-parameter was given by the source paper. Binary crossentropy calculates the error between the prediction and the actual label for a binary classification. In this particular case, the loss is calculated per log level and then combined into one loss. Note that this loss function is closely related to the sigmoid activation function in the output layer of the model. Because the output layer outputs probabilities per log level between 0 and 1, the loss function can correctly calculate the difference between the prediction and the actual label.

Other hyper-parameters are the evaluation metrics and the optimizer. The metrics will be described in subsection 6.4. The chosen optimizer is *adam*. Due to time constraints, the possible optimizers have not been properly investigated.

### 3.6 Design Decisions

The biggest design decision is concerned with the configuration and reusability of the code. At first, most design decisions were hard-coded. To make the code reusable and configurable, these configurations were later extracted from the source code. These configurations are:

- syntactic context size groups
- included file types
- included log levels
- non structural nodes
- containing block types

These configurations were not chosen at random. By tweaking these configurations, the performance of the model can be optimised. Furthermore, tweaking the configurations regarding containing block types or syntactic context size groups provides insight into the circumstances for which the model performs best.
For some configurations a mapper is created (e.g., for the included log levels). If a user decides to leave out the info level, the dataset might still contain log statements of the info level. If a log level is not included, it is mapped to a less severe level that is included. The idea behind this that certain log levels are used less frequently, or even left out in practice [19].

In the first version, the entire node around a log guard was removed. If the log guard was in an if statement, the entire if statement including its body was removed. This accidentally removed several log statements. Therefore, in later versions of the algorithm, nodes around the log guards are not just removed, but only the traces of the log guards. The removal of these log guards is described in more detail in subsection 3.1.

During preprocessing, nodes with less than 100 instances in the entire dataset are removed. The idea behind this is that these nodes only provide information for fewer than 100 log

statements, whereas these nodes add another dimension to the input vector. This threshold can be set in the configuration file for the preprocessing step. In subsection 6.5, this feature is evaluated.

# 4 Results

The code of this model can be found on GitHub. The code contains a Docker container which automatically runs the algorithm on Apache CloudStack [11]. The algorithm can also be run manually. Running the program manually consists of several steps. The first step is to manually put the source code in the download directory. By setting the *projectDirectory* variable in the configuration file, the dataset can be created. Afterwards, preprocessing, training and validation of the model is done in Python. These Python programs share one configuration file. Here too, the correct *project_directory* should be set in the configuration file. Each program outputs logs in the log directory which can be found in the dataset directory.

## 4.1 Evaluation Metrics

The metrics used in this paper are *accuracy*, *Area Under the Curve* (*AUC*) and *Avarage Ordinal Distance Score* (*AOD*). These metrics are taken from the source paper [20]. Model performance can be measured in terms of accuracy. This provides an overview of what percentage of the dataset was predicted correctly. The AUC is a more advanced metric. This metric measures the area under the *receiver operating characteristic curve* (*ROC curve*). This ROC curve plots the true positive rate against the false positive rate. The ROC curve represents the sensitivity versus the specificity of the model. The AUC represents the ROC curve in one number. The source paper states: the AUC "evaluates the ability of a model in discriminating different classes". The AUC has been slightly altered to support multiple classes [14]. The final metric used is the AOD. The AOD calculates the average distance between the predicted level and the actual level, normalized by the maximum possible distance for a certain log level [20].

Accuracy and the AUC are both implemented using keras' [6] built-in metrics. The AOD is implemented using the formula given in the source paper [20]. The source paper explicitly mentions: "accuracy in our study is the percentage of correctly suggested log levels out of all the suggestion results". For comparison, this exact definition has been implemented next tot keras' implementation. In the tables containing the results, this definition of accuracy from the source paper is used.

## 4.2 Results on Apache CloudStack

Table 4 shows the distribution of log levels throughout the features extracted from Apache CloudStack [11]. The overall performance per dataset can be found in Table 5. Lastly, Table 6 gives a more detailed overview of the evaluation of the model per feature in the dataset. In the tables regarding the evaluation of the model, some numbers are in bold. These numbers stand out as they are higher or lower than the average.

Table 4: Number of log statements per feature in Apache CloudStack

| | *trace* | *debug* | *info* | *warn* | *error* | *total* |
|---|---|---|---|---|---|---|
| *Catch Blocks* | 70 | 833 | 211 | 1154 | 1308 | 3576 |
| *For Blocks* | 11 | 16 | 13 | 8 | 4 | 52 |
| *If Blocks* | 361 | 3367 | 1113 | 1301 | 1118 | 7260 |
| *Method Blocks* | 127 | 402 | 205 | 23 | 12 | 769 |
| *Switch Blocks* | 1 | 58 | 9 | 6 | 16 | 90 |
| *Try Blocks* | 60 | 643 | 205 | 29 | 15 | 952 |
| *While Blocks* | 11 | 82 | 30 | 2 | 1 | 126 |
| $size\ '<10'$ | 240 | 914 | 357 | 207 | 190 | 1908 |
| $size\ '10-20'$ | 171 | 1513 | 445 | 745 | 722 | 3596 |
| $size\ '20-30'$ | 83 | 1001 | 310 | 576 | 562 | 2532 |
| $size\ '30-50'$ | 97 | 1209 | 323 | 625 | 602 | 2856 |
| $size\ '50-100'$ | 36 | 617 | 288 | 311 | 340 | 1592 |
| $size\ '>100'$ | 14 | 147 | 63 | 59 | 58 | 341 |

Table 5: Evaluation metrics per dataset for Apache CloudStack

| | *Accuracy* | *AUC* | *AOD* |
|---|---|---|---|
| *Training* | **94.8** | **79.8** | **97.7** |
| *Validation* | 78.1 | 73.1 | 88.7 |
| *Testing* | 77.2 | 72.8 | 88.5 |
| *All datasets combined* | 77.9 | 73.1 | 88.8 |

Table 6: Evaluation metrics per feature for Apache CloudStack

| | *Accuracy* | *AUC* | *AOD* |
|---|---|---|---|
| *Catch Blocks* | 79.4 | 71.4 | 89.0 |
| *For Blocks* | 80.8 | 73.3 | 90.7 |
| *If Blocks* | 78.0 | 72.4 | 88.6 |
| *Method Blocks* | **62.7** | **66.8** | **84.8** |
| *Switch Blocks* | 84.4 | **64.4** | 92.7 |
| *Try Blocks* | 82.6 | 71.3 | 91.4 |
| *While Blocks* | 86.5 | 72.7 | 93.8 |
| $size\ ' < 10'$ | **66.4** | 70.7 | **84.9** |
| $size\ '10 - 20'$ | 78.9 | 72.6 | 88.7 |
| $size\ '20 - 30'$ | 79.8 | 73.2 | 89.5 |
| $size\ '30 - 50'$ | 81.2 | 73.7 | 90.3 |
| $size\ '50 - 100'$ | 79.5 | 72.6 | 89.1 |
| $size\ ' > 100'$ | 83.9 | 74.6 | 92.0 |

### 4.3 Cross-System Results

The ultimate goal of log level recommendation is to perform recommendations on any type of project written in any language. Cross-system evaluation on two Apache projects is just a small step in that direction. A cross-system log level recommender should be able to extract the features necessary for log level recommendation from any project in any language. Thus, the features should be independent of the type of project or language. As described in subsection 2.3, the syntactic context abstracts away from the project-specific information. In theory, this makes the syntactic context a cross-system feature. Ideally, the model should predict log levels for systems the model has not been trained on. The exact requirements depend on the context in which the model is used.

For cross-system performance, another project from Apache is used: Apache HBase [12]. HBase is a system which can host big tables for big data. This project was also used in the source paper [20]. In the source paper, version 2.2.1 of HBase was used. For this paper, version 2.2.7 is used because version 2.2.1 is not easily accessible. In the source paper, 5.5k log statements were found, whereas in this paper 6.1k log statements are found. Although this paper does not use the same version of HBase as the source paper, the results should still resemble the results in the source paper. Table 7 shows the distribution of the log levels throughout the features extracted from the dataset. Table 8 shows the overall performance of the model per dataset and Table 9 gives a more detailed overview of the performance of the model on HBase per feature in the dataset. In the tables regarding the evaluation of the model, some numbers are in bold. These numbers stand out as they are higher or lower than the average.

Table 7: Number of log statements per feature in Apache HBase

| | *trace* | *debug* | *info* | *warn* | *error* | *total* |
|---|---|---|---|---|---|---|
| *Catch Blocks* | 39 | 128 | 65 | 579 | 544 | 1355 |
| *For Blocks* | 3 | 15 | 40 | 5 | 2 | 65 |
| *If Blocks* | 295 | 731 | 1021 | 722 | 545 | 3314 |
| *Method Blocks* | 111 | 128 | 392 | 63 | 214 | 908 |
| *Switch Blocks* | 0 | 12 | 7 | 8 | 6 | 33 |
| *Try Blocks* | 49 | 140 | 175 | 15 | 5 | 384 |
| *While Blocks* | 10 | 17 | 36 | 5 | 2 | 70 |
| $size\ ' < 10'$ | 203 | 308 | 673 | 280 | 141 | 1605 |
| $size\ '10-20'$ | 152 | 401 | 467 | 547 | 378 | 1945 |
| $size\ '20-30'$ | 89 | 218 | 273 | 297 | 441 | 1318 |
| $size\ '30-50'$ | 54 | 181 | 222 | 197 | 300 | 954 |
| $size\ '50-100'$ | 7 | 56 | 80 | 63 | 51 | 257 |
| $size\ ' > 100'$ | 2 | 7 | 21 | 13 | 7 | 50 |

Table 8: Evaluation metrics per dataset for Apache HBase

| | *Accuracy* | *AUC* | *AOD* |
|---|---|---|---|
| *Training* | **92.8** | **79.7** | **96.4** |
| *Validation* | 74.9 | 72.9 | 87.7 |
| *Testing* | 76.3 | 72.7 | 87.7 |
| *All datasets combined* | 76.2 | 72.9 | 88.2 |

cross-system performance evaluation is done in two manners. First of all, Apache CloudStack is evaluated against a model trained on both Apache CloudStack and Apache HBase. Secondly, a model trained on Apache CloudStack is evaluated on Apache HBase. The first method evaluates the model performance if the model is trained on several datasets including the dataset the model is evaluated on. The second method evaluates whether the model works on a system the model has never seen before. Table 10 and Table 11 show the performance per dataset and per feature for a model trained on a combined dataset consisting of Apache CloudStack and Apache HBase, evaluated on Apache CloudStack. Table 12 and Table 13 show the performance per dataset and per feature for a model trained on Apache CloudStack, evaluated on Apache HBase. In the tables regarding the evaluation of the model, some numbers are in bold. These numbers stand out as they are higher or lower than the average.

Table 9: Evaluation metrics per feature for Apache HBase

| | *Accuracy* | *AUC* | *AOD* |
|---:|---:|---:|---:|
| *Catch Blocks* | 75.9 | 69.1 | 89.5 |
| *For Blocks* | 80.0 | 71.0 | 88.5 |
| *If Blocks* | 76.0 | 71.6 | 88.0 |
| *Method Blocks* | 79.3 | 75.4 | 87.7 |
| *Switch Blocks* | **51.5** | **48.2** | **78.8** |
| *Try Blocks* | 74.0 | 73.5 | 86.3 |
| *While Blocks* | 77.1 | 66.6 | 88.9 |
| $size\ ' < 10'$ | 71.8 | 72.4 | 85.3 |
| $size\ '10 - 20'$ | 75.8 | 71.6 | 87.7 |
| $size\ '20 - 30'$ | 79.1 | 72.8 | 90.2 |
| $size\ '30 - 50'$ | 80.4 | 73.7 | 90.5 |
| $size\ '50 - 100'$ | 77.0 | 69.2 | 89.6 |
| $size\ ' > 100'$ | 76.0 | 67.1 | 90.7 |

Table 10: Evaluation metrics per dataset (combined dataset consisting of Apache HBase and Apache CloudStack)

| | *Accuracy* | *AUC* | *AOD* |
|---:|---:|---:|---:|
| *Training* | 76.3 | 73.0 | 88.4 |
| *Validation* | 76.6 | 72.7 | 88.3 |
| *Testing* | 75.9 | 72.0 | 88.0 |
| *All datasets combined* | 76.1 | 72.7 | 88.3 |

Table 11: Evaluation metrics per feature (combined dataset consisting of Apache HBase and Apache CloudStack)

| | *Accuracy* | *AUC* | *AOD* |
|---|---|---|---|
| *Catch Blocks* | 76.9 | 70.6 | 88.3 |
| *For Blocks* | 78.8 | 71.2 | 88.9 |
| *If Blocks* | 76.3 | 71.9 | 88.1 |
| *Method Blocks* | **60.1** | 66.6 | 84.1 |
| *Switch Blocks* | **93.3** | 67.6 | **96.4** |
| *Try Blocks* | 82.1 | 74.0 | 91.6 |
| *While Blocks* | 83.3 | **59.5** | 92.5 |
| $size~' < 10'$ | **62.8** | 70.5 | **83.7** |
| $size~'10-20'$ | 77.2 | 72.5 | 88.3 |
| $size~'20-30'$ | 79.1 | 72.6 | 89.1 |
| $size~'30-50'$ | 79.6 | 72.5 | 90.0 |
| $size~'50-100'$ | 78.1 | 71.5 | 89.4 |
| $size~' > 100'$ | 78.9 | 70.9 | 89.0 |

Table 12: Evaluation metrics per dataset (model trained on Apache CloudStack and evaluated on Apache HBase)

| | *Accuracy* | *AUC* | *AOD* |
|---|---|---|---|
| *Training* | 26.0 | 48.9 | 62.2 |
| *Validation* | 25.9 | 50.6 | 62.5 |
| *Testing* | 26.4 | 48.2 | 61.9 |
| *All datasets combined* | 26.6 | 49.4 | 62.5 |

Table 13: Evaluation metrics per feature (model trained on Apache CloudStack and evaluated on Apache HBase)

| | *Accuracy* | *AUC* | *AOD* |
|---|---|---|---|
| *Catch Blocks* | 28.6 | 40.1 | 62.7 |
| *For Blocks* | 29.2 | 44.7 | 56.4 |
| *If Blocks* | 26.0 | 42.8 | 61.7 |
| *Method Blocks* | 23.6 | **55.8** | 63.6 |
| *Switch Blocks* | 24.2 | **36.9** | 59.8 |
| *Try Blocks* | 29.9 | 43.5 | 66.2 |
| *While Blocks* | **41.4** | 44.2 | 67.5 |
| $size\ ' < 10'$ | **23.1** | 51.2 | 62.2 |
| $size\ '10 - 20'$ | 28.1 | 48.9 | 62.9 |
| $size\ '20 - 30'$ | 28.8 | 52.0 | 63.2 |
| $size\ '30 - 50'$ | 25.4 | 41.0 | 60.8 |
| $size\ '50 - 100'$ | 31.5 | 47.9 | 64.8 |
| $size\ ' > 100'$ | 28.0 | **32.6** | 61.3 |

# 5 Responsible Research

In this section, the ethical implications of the research are discussed. This section is closely related to section 6. In this section, the focus is on the ethical implications and limitations of the approach in this paper, whereas section 6 focuses on the optimality of the model.

## 5.1 The Dataset

The dataset in this paper is composed of Apache CloudStack [11]. The decision to use this project was made by the supervisor. Many projects rely on Apache CloudStack. If the quality of the logs from Apache CloudStack would be low, these logs would pollute the logs of the projects depending on Apache CloudStack. Therefore, this paper is based on the assumption that the logging quality of Apache CloudStack must be of a high standard.

For the results in this paper, it is essential that the usage of log levels in regard to their syntactic context is consistent throughout Apache CloudStack. As described in subsection 6.2, the relation between the log level and the syntactic context cannot be easily articulated in a small set of rules, because of which it is hard to check whether the dataset is consistent. Noise, in the form of inconsistent log levels regarding their syntactic context, would decrease the precision of the model.

Next to the above mentioned limitations, there are several more limitations to this dataset. The dataset represents only one type of system (cloud computing). Furthermore, the entire

system is written in Java. Besides, the dataset is relatively small, as can be seen in Table 2, Table 3 and Table 4. Overall, there are multiple serious limitations to the dataset. The aim of this paper is not to create a model for general usage. The aim of this paper is to confirm the promising results of automated deep learning based on syntactic context features for log level recommendation. The limitations do influence the results and should not be overlooked, but these limitations do not invalidate this research.

## 5.2 Reproducibility

Dependencies influence the reproducibility of code [7]. The model in this paper does depend on several external Java and Python dependencies. All of these dependencies form a threat to the validity of the model. There might be a flaw in any of these dependencies that causes the model to fail. Furthermore, to reproduce the results in this paper, the entire environment should be equal, including all dependencies. The best solution to this problem would be having no external dependencies. Another option would be to thoroughly investigate external dependencies. Both solutions have not been done in this paper. For this paper, there is a serious time constraint. Above that, the aim of this paper is not to invent a new way of performing log level recommendation, but to validate the results from another paper. Therefore, it would have been better if the model did not depend on external dependencies, but that is not necessary for this paper. The dependencies used in this paper are open source. The dependencies used are common in the field. This does not prove that the library does not contain any bugs, but it is an indication that the library is trustworthy. Still, these dependencies should not be overlooked, and form a limitation for the model in this paper.

Certain dependencies used by the model in this paper are licensed under the *GNU General Public License* [13]. As the license states, upon usage of licensed software “you must pass on to the recipients the same freedoms that you received”. As a consequence, recipients of the software described in this paper have the same freedoms as described in the GNU General Public License [13].

In order to make this research reproducible, the source code of this paper contains a Docker container [1]. The Docker image creates a reproducible environment “including all dependencies, configuration, code and data needed” [7]. The dependencies used for the model in this paper are: JavaParser [18], JSON [23], CK [2], Pandas [28], Numpy [15], Python-tabulate [3], Tensorflow [21], and Keras [6].

## 5.3 The Model

The model as described in this paper can result in false positives or false negatives. The severity of such a wrong recommendation depends on the context. Users of this model should acknowledge the limitations of this study. To begin with, this model performs recommendations. The recommendation done by the model should under no circumstances be interpreted as ‘the truth’. Users of the model should be aware of eventual differences between their specific use case and the code used during training. Use cases might differ among others in programming language, type of application, or log level standards. Furthermore, this model was not intended for general usage, but as validation of a paper published on log level recommendation [20]. As a result, important decisions, for instance which dataset to use, were made regarding this objective, and not for general usage.

# 6 Discussion

In this section, the optimality of the model in this paper is debated and compared to alternatives. This section is closely related to section 5 where the ethical implications of the research are discussed.

## 6.1 The Dataset

This subsection focuses on the distributions of the different features in Apache CloudStack. Afterwards, subsection 6.5 describes the influence of these features on the model performance.

As Li et al. [19] mention, certain log levels are used more frequently than others. Similar to their findings, Apache CloudStack [11] does not contain any log statements of level fatal. Next to fatal, trace is used the least. This is again in line with the findings by Li et al..

As mentioned in subsection 2.3, the context of a log statement is believed to be a strong indicator for log level recommendation. In both papers, these contexts are divided into several categories called *containing block types*. In this paper, the containing block types from the paper by Li et al. [19] are used. As opposed to Li et al., most containing blocks have an equal distribution of log levels in this paper (Table 4). All containing block types except Catch Blocks have most of their log statements in either debug or info. For these blocks, debug and info form the center of a distribution approximately equal to a normal distribution. As mentioned, Catch Blocks form an exception here. For Catch Blocks the exception is that warn and error break with this pattern from the other blocks. The interesting bit here is that this holds for both Apache CloudStack [11] and Apache HBase [12] (Table 4 and Table 7). The exceptional behaviour of Catch Blocks is not really exceptional behaviour. The behaviour of code in a catch block is expected to be exceptional. As a consequence, the log levels in these blocks tend to be more severe than the log levels in other blocks. However, as mentioned in subsection 6.2, the distributions in the tables also indicate that log level recommendation is not as easy as stating *if the log statement is in a catch block, the log level is error* [19]. For both projects, Catch Blocks contain a significant amount of log statements with log level debug.

As introduced in subsection 3.1, the size of the context is saved to evaluate the performance of the model regarding different syntactic context sizes. Subsection 6.5 describes the influence of the syntactic context size on performance. The different syntactic context sizes were picked by manual investigation into the different syntactic context sizes in the dataset. Tweaking those syntactic context size groups might result in more insight in what circumstances make the model perform optimally.

## 6.2 Deep Learning

Is machine learning (ML) the best approach for log level recommendation? Amazon Web Services, Inc. or its affiliates provide two practical guidelines for this question [16]. As the first guideline, the guide mentions that ML works best for problems that “cannot be adequately solved using a simple (deterministic), rule-based solution”. As mentioned above, looking at the log level recommendation problem, most research indicates that it is not enough to make simple rules like: *if the log statement is in a catch block, the log level is*

*error* [19] [20]. Many factors influence the decision upon log level. Furthermore, several of these factors overlap. By intuition, it makes a huge difference whether a Catch Block captures a critical failure of an ATM or that the Catch Block catches a *User not found* exception for a login trial. Although containing block types are a good metric, it is not precise enough to base the entire recommendation process solely on this metric. In short, the log level recommendation problem cannot be captured in a small set of rules. The other guideline stated by the guide describes the scale of the problem: "ML solutions are effective at handling large-scale problems" [16]. For log level recommendation, manually recommending the log level for millions of log statements becomes tedious. In conclusion, ML seems to be a good approach for log level recommendation.

The effectiveness of a Machine-Learning model mostly depends on "the nature and characteristics of the data, and the performance of the learning algorithms" [27]. As subsection 5.1 explains, this paper assumes that Apache CloudStack [11] provides a consistent dataset regarding the relation between the log levels and their corresponding syntactic context. The effectiveness of a Machine-Learning model also dependeds on the performance of the learning algorithm. Deep learning mostly outperforms general ML algorithms when the dataset is large [27]. Regarding log levels, there is a lot of data to be mined from open source projects, which makes deep learning a good option for the log level recommendation problem. In particular, a RNN is "well-suited for analyzing and learning sequential data" [27]. A RNN fits well with the syntactic context metric as the order of the AST nodes in the syntactic context is important, as described in subsection 2.3.

However, the optimality of the approach described in this paper should not be exaggerated. The combination of a RNN with the syntactic context feature using ordinal encoding for the log levels seems to be a strong combination. It is however not proven that this approach is the optimal approach for log recommendation, other methods might outperform this model. Nevertheless, based on the current knowledge, a RNN in combination with the syntactic context feature using ordinal encoding for the log levels is the best method for log level recommendation.

### 6.3 Bi-LSTM

The specific algorithm used in this paper is Bi-LSTM as this algorithm can capture the order of the context in the input vectors. As subsection 2.3 explains, this algorithm captures the control flow in the source code. However, the source paper [20] does not go into detail on how this ordering is kept throughout embedding (subsection 3.2). The syntactic context consist of node types. These node types are converted into frequency vectors, which are then embedded. The ordering of the original syntactic context vector is gone after embedding as the conversion to frequency vector, as described in subsection 3.2, does not maintain the order of the original vector. The source code from the source paper is not available. As a result, it is hard to determine how the authors preserved the order of the syntactic context while converting the vectors to frequency vectors. At one point the source paper mentions that the features are based on basic blocks [20]. This might indicate that the syntactic context should be represented as a vector of vectors, one vector per basic block in the syntactic context. This way, the ordering of the basic blocks is kept during the conversion to frequency vectors. For this paper, due to time constraints, this has not been investigated.

## 6.4 Evaluation Metrics

Do the chosen evaluation metrics give a good indication of how well the model performs in log level recommendation? Evaluation is a critical part of the ML pipeline. If the evaluation part of the model is incorrect, the model will be trained and optimised in the wrong direction. Japkowicz and Shah (2011) describe this in their book "Evaluating Learning Algorithms : A Classification Perspective" [17]. As they mention, the evaluation is often used as a routine, without properly thinking about the specific requirements for a given algorithm. It is of critical importance to use evaluation metrics that evaluate specific requirements of the model. For the model described in this paper, the requirement is rather general: the model should accurately recommend log levels. Because this paper only validates another paper, there are no other, more specific requirements. For this paper, the evaluation metrics are based on the source paper [20]. As mentioned in subsection 4.1, these metrics are accuracy, AUC, and AOD. These metrics indicate three things:

1. The percentage of correctly recommended log levels (Accuracy)
2. The ability to discriminate between different log levels (AUC)
3. The average distance between the wrong recommendations and the actual log levels (AOD)

Combined, these metrics seem to give a detailed overview of how well the model can recommend log levels in a general way, both for within-system evaluation and cross-system evaluation. On the other hand, the correctness of the model performance is very context dependent. For specific projects, it might be critical that certain log statements have severe log levels. These projects have very specific requirements. The metrics in this paper result in an indication of correctness for general usage, and thus does not encapsulate any specific requirements. This is a serious restraint on the concept validated in this paper. The results in this paper cannot be generalised to any specific scenario. Specific requirements might invalidate the results in this paper. Therefore, the metrics in this paper cover the requirements for this specific paper. For any other use case, other metrics might apply that encapsulate different requirements.

## 6.5 Results

The last subsection of the discussion discusses the results obtained in this paper. In this subsection several interesting details of the results are worked out. This subsection indicates what is interesting in the results, but does not always answer the question of what is the cause for a certain point of interest in the dataset. This is future work.

To begin with, as this paper is a replication of the paper by Li et al. [20], the results in this paper should be compared to the results in the source paper. The accuracy is far apart as compared to the source paper. The average accuracy in the source paper is 54, whereas in this paper the accuracy is close to 75. As mentioned in subsection 4.1, the accuracy metric used in this paper is directly based on the accuracy metric described in the source paper. Still, the most plausible explanation for this difference seems to be a difference in evaluation metrics. This has not been investigated.
The AUC in this paper is slightly lower than the average AUC in the source paper. The source paper has an average AUC of 80.8 whereas this paper has an average AUC around 72. The built in metric of keras [6] (subsection 4.1) might slightly differ from the implementation

used in the source paper. As discussed in subsection 6.3, the source paper does not provide much detail on the feature extraction part of the algorithm. Their approach might better capture the order of the tokens in the syntactic context, which could explain the difference in performance regarding the AUC. Both options have not been investigated.
Lastly, the AOD in this paper (approximately 88) is higher than the AOD in the source paper (average 79.3). The implementation of this metric was taken from the source paper [20] (subsection 4.1). A possible explanation might be that the projects used for the models differ. However, with this assumption, the AOD on HBase should be roughly equal to the AOD on HBase in the source paper (81.4 in the source paper vs 87 in this paper). Another explanation might be that the extraction method differs, as described in subsection 6.3. This difference might be the cause for the difference in AOD. These possible reasons have not been investigated.

Regarding the the performance per dataset, the model is clearly overfitted. This can be seen in Table 5. The model performs better on the training set than on any other dataset. The performance on the other datasets (total set, validation set and testing set) is more stable. Apache HBase [12] and Apache CloudStack [11] show the same overfitting (compare Table 5 and Table 8). This could mean several things. First of all, the dataset might contain inconsistent logging practices. As subsection 5.1 describes, this paper depends on the assumption that CloudStacks quality of logging is of a high standard. This could also mean that the log level recommendation problem is so complicated, that the dataset is too small to capture all syntactic context patterns. Another possible reason for the overfitting is the model itself. The model is a replication of the model described by Li et al. [20]. The paper mentions that the model includes a "0.2 dropout rate, to reduce the potential impact of overfitting on the trained system". This has been adopted in the model in this paper. Still, the result is strongly overfitted. This might indicate an implementation mistake in this paper, but this might also mean that the dropout rate did not prevent overfitting.

Table 6 shows that the performance regarding the containing block types is fluctuating a lot. This seems to be project-dependent. For CloudStack, performance is worst for Method Blocks whereas in HBase, performance is worst for Switch Blocks. Although there is not a lot of data for these specific block types (Table 4), this paper does not indicate a direct relation between the amount of data for a certain feature and the model performance on that feature. For CloudStack, there is little data on For Blocks, but the model performs well on this feature. On the other hand, there is a lot of data on If Blocks, while the model performance is not that good on this specific block type. This fluctuation in performance can have several reasons. First of all, the containing block types feature has not been included in the stratified sampling. In other words, the distribution of the containing block types is random in the training set, and might be imbalanced regarding the distribution in the entire dataset (subsection 3.1). Furthermore, there is only little data on certain features. For the features for which there is a lot of data, the performance converges to the overall performance of the model. This fluctuation regarding the containing block type might be solved by expanding the dataset with more samples on sparse features. The containing block types should than be included during stratified sampling. However, this fluctuation might also indicate that the model is not optimal for certain features. This can however not be concluded based solely on the results in this paper. Further investigation is necessary before such conclusions can be drawn.

Table 14: Evaluation metrics per dataset for Apache CloudStack (no frequency threshold)

| | *Accuracy* | *AUC* | *AOD* |
|---:|---:|---:|---:|
| *Training* | 94.8 -> 94.2 | 79.8 -> 79.8 | 97.7 -> 97.4 |
| *Validation* | 78.1 -> 75.4 | 73.1 -> 72.7 | 88.7 -> 88.1 |
| *Testing* | 77.2 -> 76.9 | **72.8 -> 73.1** | 88.5 -> 88.5 |
| *All datasets combined* | 77.9 -> 76.9 | 73.1 -> 72.9 | 88.8 -> 88.5 |

In Table 6, the model performance is shown with regard to the syntactic context size groups (subsection 3.1). As subsection 6.1 described, these groups were manually chosen and not optimised in terms of finding the optimal boundaries for which the model performs differs. Overall the table seems to indicate that the model performs better on large syntactic contexts. Table 9 partly coincides with this. Here the model trained on HBase performs best on the middle groups. One detail to not be overlooked is that HBase contains only little data on large syntactic context size groups (Table 7). Identical to the containing block types, the syntactic context size groups have not been included during stratified sampling (subsection 3.1). Focussing on the smaller syntactic context size groups, for which there is enough data in both projects, there seems to be a relation between the size of the syntactic context and the model performance. Opposed to the containing block types, this relation does not seem to be project-dependent.

As mentioned in subsection 3.6, nodes with less than 100 instances are removed from the dataset. In order to validate whether this threshold improves the performance of the model, the model was also run with a threshold of 0. In other words, all nodes were kept. The results are visible in Table 14 and Table 15. The entries in bold represent entries for which the performance improved by lowering the threshold from 100 to 0. As can be seen in the tables, for most entries there is only a minor increase or decrease. For the containing block types, some entries show a large increases or decreases. As mentioned earlier in this subsection, the performance per containing block type is less stable. Once again, it is important to mention that for some of these containing block types, there is only little data. The population distribution of these containing block types has not been included during random sampling. As a consequence, the increase or decrease in performance for certain containing block types could be caused by the change in threshold, but this might also be because of an accidental imbalance in the training data (subsection 3.1). Table 14 shows that overall, the performance seems to slightly decrease because of the change in threshold from 100 to 0. In conclusion, the decision to include a threshold for nodes with a low frequency seems to have increased the performance of the model. The optimal threshold has not been investigated.

In order to investigate whether the performance improves using fewer log levels, this paragraph compares a model using 5 log levels with a model using only the four middle log levels (debug, info, warn, and error). As described in subsection 3.6, which log levels to include can be configured. As can be seen in Table 3, Apache CloudStack contains no fatal log statements. As subsection 6.1 mentioned, trace is another level that is not frequently used, just as Li et al. [19] found. Table 16 and Table 17 show that overall, using 4 log levels,

Table 15: Evaluation metrics per feature for Apache CloudStack (no frequency threshold)

| | *Accuracy* | *AUC* | *AOD* |
|---|---|---|---|
| *Catch Blocks* | 79.4 -> 77.6 | **71.4 -> 71.6** | 89.0 -> 88.4 |
| *For Blocks* | 80.8 -> 76.9 | 73.3 -> 68.2 | 90.7 -> 87.5 |
| *If Blocks* | 78.0 -> 77.1 | 72.4 -> 72.0 | 88.6 -> 88.4 |
| *Method Blocks* | **62.7 -> 63.6** | 66.8 -> 64.7 | 84.8 -> 85.1 |
| *Switch Blocks* | 84.4 -> 83.3 | **64.4 -> 76.4** | 92.7 -> 91.2 |
| *Try Blocks* | 82.6 -> 83.0 | 71.3 -> 71.8 | 91.4 -> 92.1 |
| *While Blocks* | 86.5 -> 80.2 | 72.7 -> 60.3 | 93.8 -> 89.7 |
| $size\ ' < 10'$ | 66.4 -> 64.6 | 70.7 -> 70.6 | 84.9 -> 83.7 |
| $size\ '10 - 20'$ | 78.9 -> 78.4 | 72.6 -> 72.4 | **88.7 -> 88.9** |
| $size\ '20 - 30'$ | 79.8 -> 78.9 | 73.2 -> 73.1 | 89.5 -> 89.4 |
| $size\ '30 - 50'$ | 81.2 -> 79.0 | 73.7 -> 72.9 | 90.3 -> 89.1 |
| $size\ '50 - 100'$ | **79.5 -> 80.5** | 72.6 -> 71.9 | **89.1 -> 90.1** |
| $size\ ' > 100'$ | 83.9 -> 81.2 | 74.6 -> 72.6 | 92.0 -> 90.9 |

the accuracy improves, the AUC decreases while the AOD stays the same. The bold entries represent the entries that have increased due to the change from 5 to 4 log levels. 4 log levels improves performance for some specific features. For instance the Switch Block performs better using 4 log levels. For other features, performance drops due to the 4 log levels, for instance syntactic context size group '> 100'. 4 log levels seems to increase performance for the smaller syntactic context sizes. Specific configurations per feature could improve the model (e.g., Switch Blocks use 4 log levels whereas size group '> 100' uses 5 log levels). As mentioned above, the containing block type was not included in the stratified sampling process. The performance drop or improvement for the containing block types might be due to the change in log levels, but this might also be the consequence of an accidental sampling imbalance regarding the containing block types (subsection 3.1). This has not been investigated.
The decrease in AUC was not expected. Why would it be harder to discriminate between 4 log levels than for 5 log levels? A possible explanation is that this indicates the different use case scenarios for the different log levels. In this study, when a certain log level is not included, it is mapped to the closest log level during feature extraction. For the specific scenario in this paragraph, all trace log statements were mapped to debug during feature extraction. This has been done because of the assumption that users would still want to log the place containing a trace statement if log level trace would not exist. Performance might increase if these trace statements were ignored instead of mapped. Above that, the performance might also increase when the mapping would be done after training the model instead of during feature extraction. Both possible improvements have not been investigated.

The final topic for discussion is how the model performs in cross-system validation as intro-

Table 16: Evaluation metrics per dataset for Apache CloudStack (4 log levels)

| | *Accuracy* | *AUC* | *AOD* |
|---:|---:|---:|---:|
| *Training* | **94.8 -> 95.9** | 79.8 -> 74.9 | 97.7 -> 97.6 |
| *Validation* | **78.1 -> 80.7** | 73.1 -> 69.2 | 88.7 -> 88.1 |
| *Testing* | **77.2 -> 78.6** | 72.8 -> 68.3 | 88.5 -> 86.6 |
| *All datasets combined* | **77.9 -> 80.2** | 73.1 -> 69.1 | 88.8 -> 87.6 |

Table 17: Evaluation metrics per feature for Apache CloudStack (4 log levels)

| | *Accuracy* | *AUC* | *AOD* |
|---:|---:|---:|---:|
| *Catch Blocks* | 79.4 -> 78.3 | 71.4 -> 67.5 | 89.0 -> 86.1 |
| *For Blocks* | **80.8 -> 84.6** | 73.3 -> 63.4 | 90.7 -> 89.7 |
| *If Blocks* | **78.0 -> 80.4** | 72.4 -> 68.3 | 88.6 -> 87.6 |
| *Method Blocks* | **62.7 -> 76.1** | 66.8 -> 61.6 | **84.8 -> 87.4** |
| *Switch Blocks* | **84.4 -> 91.1** | **64.4 -> 72.8** | **92.7 -> 94.4** |
| *Try Blocks* | **82.6 -> 86.7** | 71.3 -> 68.4 | **91.4 -> 92.2** |
| *While Blocks* | **86.5 -> 90.5** | 72.7 -> 71.4 | **93.8 -> 96.0** |
| $size' < 10'$ | **66.4 -> 77.0** | 70.7 -> 68.6 | **84.9 -> 86.5** |
| $size'10-20'$ | **78.9 -> 80.9** | 72.6 -> 69.5 | 88.7 -> 88.0 |
| $size'20-30'$ | **79.8 -> 81.5** | 73.2 -> 69.2 | 89.5 -> 87.9 |
| $size'30-50'$ | 81.2 -> 81.2 | 73.7 -> 69.0 | 90.3 -> 88.0 |
| $size'50-100'$ | 79.5 -> 79.3 | 72.6 -> 68.6 | 89.1 -> 87.5 |
| $size' > 100'$ | 83.9 -> 77.4 | 74.6 -> 67.2 | 92.0 -> 86.5 |

duced in subsection 4.3. The source paper shows promising results regarding cross-system validation [20]. In the source paper, the approach is to combine data from all projects studied in the paper and form one combined dataset from the log statements in those projects. The results of validating a model trained on this combined dataset are promising. In this paper, this approach is replicated, but also extended. Whereas the source paper validates that the algorithm still outperforms all baselines if the dataset is not solely based on the project it is validated on, this paper also investigates what happens with the performance once the model is trained on a different project than it is evaluated on.

As Table 10 shows, the performance on a combined dataset is very stable and almost equal to within-system performance (Table 5). In the source paper, cross-system performance is measured proportional against within-system performance. The accuracy drops significantly compared to within-system performance (average of 83.1%), the AUC drops with 8% (average of 91.8%) and the AOD stays almost the same (average of 96%). In this paper, the accuracy also drops most, but the difference is smaller (97% when taking an average of 78 for within-system performance and an average of 76 for cross-system performance). The AUC and AOD do not really change. Note that this does not hold for the training set. However, as mentioned above, the training seems to be overfitted. Whereas the source paper concludes that cross-system evaluation gives encouraging results, this paper can conclude that cross-system evaluation gives nearly the same results as within-system evaluation. Regarding the features, Table 11 demonstrates an almost equal distribution of metrics as the within-system performance evaluation (Table 6). In both tables, the Method Blocks and Switch Blocks perform differently than the other features. Here too, Method Blocks perform worse than average for any metric whereas Switch Blocks perform worse for the AUC but better than average in terms of accuracy and AOD. Table 11 also indicates that the smallest syntactic context size group performs worst. Here too, the model performs better on larger syntactic context size groups, with optimal performance for the middle sizes (size group 30 - 50). This resembles the behaviour as discussed in the paragraph on the syntactic context size feature.

Table 12 and Table 13 visualize the performance drop once the model is evaluated on a dataset the model has never seen before. The performance distribution is still somewhat similar to the distribution in Table 5, where this model was evaluated on CloudStack instead of HBase. However, the accuracy of random guessing using 5 log levels is around 20. Regarding the AUC, "an AUC lower than 0.5 indicates a performance that is not better than random guessing" [20]. The main message here is that once the model is evaluated on a different system than the system the model is trained on, then the model performance is only slightly better than a random guessing model.

The above discussed model performance regarding cross-system evaluation mean that the model can only be used for projects the model has been trained on. This means that this model can only be used for projects that already have a sufficient amount of logging statements to train the model. This also means that the model cannot be used on projects that do not contain a lot of logging statements yet.

The cross-system performance results reinforce the notion that different systems have different logging practices. The results in Table 12 and Table 13 put this research into perspective: currently it is possible to recommend log levels on a dataset the model equal to the dataset

the model was trained on. This dataset can even be expanded with data from other projects (Table 10 and Table 10). However, the model is not at a stage yet that the model can accurately predict the log levels for a system the model has never seen before.

# 7 Conclusions and Future Work

This paper tries to validate the results of another paper [20] on automated log level recommendation. This paper answers two research questions related to the research in the source paper. These research questions are:

- RQ1: What is the performance of automated deep learning based on syntactic context features for log level recommendation?
- RQ2: Is the approach able to perform cross-system recommendations?

With regard to the first research question, although the evaluations in this paper are sometimes far apart from the evaluations in the source paper, the results still validate the concept proposed in the source paper. The evaluations in section 4 show that automated deep learning based on syntactic context features for log level recommendation can predict log levels with an accuracy of 78, an AUC of 73, and an AOD of 88. As the source paper describes, these results validate that the syntactic context (subsection 2.3) is a good metric for log level recommendation and that the ordinal nature (subsection 2.2) of the log levels is important to include in the recommendation process [20].

For the second research question, this paper executed two evaluations: one evaluation on a model that was trained on several datasets including the dataset the model was evaluated on, and one evaluation on a model that was trained on another dataset than the dataset the model was evaluated on. The performance of a model on a combined dataset is almost equal to within-system performance, as described in subsection 6.5. However, once the model is evaluated on a system that the model has never seen before, performance drops to nearly random guessing. In conclusion, the model is able to perform log level recommendations as long as the data for which the model predicts the log level was represented in the dataset the model was trained on.

Based on the findings in this paper, several features of the model seem promising for future research. The containing block types could be further investigated. A possible improvement could be to use custom configurations per containing block type. These configurations should exploit the different ways these containing block types are used with regard to log levels [19]. The syntactic context size seems to be an important factor for log level recommendation. Future research could investigate the optimal syntactic context sizes for which the model performs best. For both the containing block types and the syntactic context sizes, one key requirement is that the dataset contains enough data on the different block types or syntactic context size groups (subsection 6.5). These different block types or syntactic context size groups should then be included during stratified sampling, as described in subsection 3.1 and subsection 6.5. Another interesting finding in this paper is that the different nodes in the contexts should be filtered based on their frequency. As subsection 6.5 discusses, the model seems to perform better while certain nodes are filtered from the syntactic context based on their frequency in the overall dataset. The optimal threshold should be found for which the model performs best. This idea could be worked out further. What if several

different nodes are mapped to one overlapping node type (e.g., mapping both *ForStmt* and *ForeachStmt* to *ForStmt*)? Doing so could teach the model similarities between certain node types. Next to that, this would decrease the size of the input vector and potentially increase model performance.

The model in this paper is not perfect. As stated in subsection 5.1, this paper is based on the assumption that the logging quality of Apache CloudStack is of a high standard. As mentioned several times throughout this paper and discussed in section 6, certain hyper-parameters and configurations could be optimised in future research. There seems to be a problem with the feature extraction step in this paper as the current algorithm does not keep the order of the context, as described in subsection 6.3. The source paper does not reveal a lot of implementation details on this part of the algorithm. Still, this step of the model should be improved such that the order of the context is kept, even through the translation from syntactic context nodes to frequency vectors. This might even improve the results of the algorithm to more closely match the results in the source paper (subsection 6.5).

In a more general sense, this model could be improved by capturing not only the syntactic context of a log statement, but also the context of the system around the log statement. As mentioned in subsection 5.3, the severity of a wrong recommendation depends on the context of a system. If a future version of the algorithm described in this paper would be able to capture the context of the system and adapt the logging practices accordingly, this could lead to drastic performance increases. Furthermore, further research can be done regarding the chosen metrics. In this paper the chosen methodology is to hand the syntactic context over to the model and let the model find the necessary relations in the syntactic context. However, as both papers mentioned in section 1 show, humans have their own methods of interpretation for the syntactic context. Combining human intelligence with artificial intelligence could result in better performance.

As a final remark, this paper is just a very small step in an enormous field: automated log level recommendation. In this paper validates the possibility to recommend log levels based on the syntactic context using data from Apache CloudStack [11] to predict log levels for Apache CloudStack. As an extra feature, cross-system performance is evaluated on another Apache project. By doing so, this paper validates that the methodology described in the paper by Li et al. [20] shows potential for further research. This paper does not validate that the model is production ready. This paper rather encourages to put more research into the topic. The model proposed by Li et al. [20] could be one of the first steps towards a model that is capable log level recommendation for any system, written in any language, in any context.

# References


[1] Charles Anderson. "Docker [Software engineering]". In: *IEEE software* 32.3 (2015), pp. 102–c3. ISSN: 0740-7459.

[2] Maurício Finavaro Aniche. *CK*. Version 0.6.4. 2021. URL: `https://github.com/mauricioaniche/ck` (visited on June 1, 2021).

[3] Sergey Astanin et al. *python-tabulate*. Version 0.8.9. Feb. 2021. URL: `https://pypi.org/project/tabulate/` (visited on June 10, 2021).

[4] Jeanderson Cândido et al. "An Exploratory Study of Log Placement Recommendation in an Enterprise System". In: *CoRR* abs/2103.01755 (2021). arXiv: 2103.01755.

[5] Jeanderson Barros Cândido, Maurício Finavaro Aniche, and Arie van Deursen. *Log-based software monitoring: a systematic mapping study*. 2021. arXiv: 1912.05878 [cs.SE].

[6] François Chollet et al. *Keras*. 2015. URL: https://keras.io (visited on June 1, 2021).

[7] J. Cito and H. C. Gall. "Using Docker Containers to Improve Reproducibility in Software Engineering Research". In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE Computer Society, May 2016, pp. 906–907. URL: https://doi.ieeecomputersociety.org/.

[8] William Gemmell Cochran. *Sampling techniques*. eng. 3rd ed. Wiley series in probability and mathematical statistics 810651580. New York ; London [etc.]: Wiley, 1977. ISBN: 047116240X.

[9] Georgios Drakos. *What is an Embedding Layer?* June 2019. URL: https://gdcoder.com/what-is-an-embedding-layer/ (visited on June 16, 2021).

[10] Apache Software Foundation. *Apache Log4j*. Version 2.14.1. 2021. URL: https://logging.apache.org/log4j/2.x/ (visited on May 28, 2021).

[11] Apache Software Foundation. *CloudStack*. Version 4.15.0.0. 2020. URL: https://cloudstack.apache.org/ (visited on May 27, 2021).

[12] Apache Software Foundation. *HBase*. Version 2.2.7. Apr. 2021. URL: https://hbase.apache.org/ (visited on June 17, 2021).

[13] *GNU General Public License*. Version 3. Free Software Foundation, June 29, 2007. URL: http://www.gnu.org/licenses/gpl.html (visited on June 1, 2021).

[14] David J Hand and Robert J Till. "A Simple Generalisation of the Area Under the ROC Curve for Multiple Class Classification Problems". eng. In: *Machine learning* 45.2 (2001), pp. 171–186. ISSN: 0885-6125.

[15] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[16] Amazon Web Services Inc. or its affiliates. *When to Use Machine Learning*. 2021. URL: https://docs.aws.amazon.com/machine-learning/latest/dg/when-to-use-machine-learning.html (visited on May 27, 2021).

[17] Nathalie Japkowicz. *Evaluating learning algorithms a classification perspective*. eng. Cambridge [England] ; New York: Cambridge University Press, 2011. ISBN: 1-107-21465-3.

[18] JavaParser.org. *JavaParser*. Version 3.22.0. 2019. URL: https://javaparser.org/ (visited on June 1, 2021).

[19] Heng Li, Weiyi Shang, and Ahmed E Hassan. "Which log level should developers choose for a new logging statement?" In: *Empirical software engineering : an international journal* 22.4 (2017), pp. 1684–1716.

[20] Zhenhao Li et al. "DeepLV: Suggesting Log Levels Using Ordinal Based Neural Networks". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2021, pp. 1461–1472. DOI: 10.1109/ICSE43902.2021.00131.

[21] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/` (visited on June 1, 2021).

[22] Tomas Mikolov et al. "Efficient Estimation of Word Representations in Vector Space". eng. In: (2013). URL: `http://arxiv.org/abs/1301.3781`.

[23] org.json. *json*. Version 20210307. 2021. URL: `https://search.maven.org/artifact/org.json/json/20210307/bundle` (visited on June 1, 2021).

[24] Heidar Pirzadeh et al. "Stratified sampling of execution traces: Execution phases serving as strata". eng. In: *Science of computer programming* 78.8 (2013), pp. 1099–1118. ISSN: 0167-6423.

[25] Rutger Ruizendaal. *Why You Need to Start Using Embedding Layers*. July 2017. URL: `https://towardsdatascience.com/deep-learning-4-embedding-layers-f9a02d55ac12` (visited on June 16, 2021).

[26] Stuart Russell. *Artificial intelligence : a modern approach*. eng. 2nd [ext. and rev.] ed. Prentice Hall series in artificial intelligence 138675007. Upper Saddle River, NJ [etc.]: Prentice Hall/Pearson Education, 2003. ISBN: 0137903952.

[27] Iqbal H Sarker. "Machine Learning: Algorithms, Real-World Applications and Research Directions". In: *SN computer science* 2.3 (2021), pp. 160–160.

[28] The pandas development team. *pandas-dev/pandas: Pandas*. Version 1.2.4. Feb. 2020. DOI: `10.5281/zenodo.3509134`. URL: `https://doi.org/10.5281/zenodo.3509134` (visited on June 1, 2021).

[29] Jian Zhang et al. "A Novel Neural Source Code Representation Based on Abstract Syntax Tree". In: ICSE '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 783–794. DOI: `10.1109/ICSE.2019.00086`. URL: `https://doi.org/10.1109/ICSE.2019.00086`.