Machine Learning Approach to 5G Layer 1 Code Review

Michał Porębski

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 30.11.2022

Supervisor

Prof. Esa Kallio

Advisor

MSc Marko Tuononen

Aalto University
School of Electrical
Engineering

Aalto University
School of Electrical
Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Michał Porębski

Title Machine Learning Approach to 5G Layer 1 Code Review

Degree programme Electronics and electrical engineering

Major Space Science and Technology **Code of major** ELEC3039

Supervisor Prof. Esa Kallio

Advisor MSc Marko Tuononen

Date 30.11.2022 **Number of pages** 54 **Language** English

Abstract

The programming is used in most of the industries and domains of life. Programming projects are becoming bigger and bigger, with millions of developers working on them across the world. Such projects are sometimes the core of precise, delicate and expensive operations, like space missions. They often require autonomous work for many years, therefore they have to be thoroughly tested before the exploitation. Hence, each change which is done in such project needs to be verified by automatic system and other programmers. It is not a trivial task, because a typo, a bug, a security violation, etc. easily appear in the billions of lines of code. Such mistakes need to be found and fixed, otherwise the consequences can be devastating. For that purpose, many automatic bug finding approaches are being researched. The deep neural networks are the most promising solutions. They allow for checking the issues which were caught only by other programmers and not by already existing automatic systems.

This work focuses on machine learning approach to code review and software quality assurance. It describes the recreation of neural network deepreview model and experiments with its modifications. It also proposes a different approach to a feature extraction phase. The thesis consists of descriptions of created architectures, shows results of the experiments and compares them with original article. The implemented models are tested on the database gathered from specific branch of Nokia Corporation responsible for implementation of 5G layer 1. It is described how such data are processed and analysed. It also provides a short history of the evolution of such automatic systems for code review.

Keywords machine learning, neural networks, software quality assurance, code review

Preface

I would like to express my sincere gratitude to my advisor, Marko Tuononen, for providing the good and stubborn guidance through the whole project and for having the patience for my slow problem-solving.

I want to extend my appreciation to Professor Esa Kallio for having the patience for me and good guidance along the way.

I would like to thank my fiancé, Klaudii for keeping me going against many adversities and always being there for me when I needed it.

I highly appreciate the support of my parents, brother, his wife and my friends. The time and attention they gave me were priceless.

I want to thank my team members and my manager, Topi Rantalainen for listening to my stuttering during daily meetings.

At the end, I would also like to thank my flatmate, Jakubowi Cisło, for having constant doubts that I will ever finish this thesis.

The thesis was ordered and supervised by Nokia Corporation.

Espoo, 30.11.2022 Michał Porębski

Contents

Symbols and abbreviations

ADC	Analog to digital converter
API	Application Programming Interface
AUC	Area under the curve
BOW	Bag-of-words
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DAC	Digital to analog converter
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit Neural Network layer
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
LSTM	Long Short-Term Memory Neural Network layer
NN	Neural Network
nD	n-dimensional
RNN	Recurrent Neural Network
ROC	Receiver operating characteristic
TFIDF	Term frequency–inverse document frequency

1 Introduction

In commercial programming projects, the amount of code and commits with changes is enormous. Each project repository usually requires the work of tens of developers. Every time, the change is being done it must be approved and reviewed by other programmers before it can be merged into the production version. There also exists projects which require autonomous work through many years, e.g., space missions. Such ventures require thorough testing on every step, because the bugs and possible failure points in the code are very well hidden. Although, systems for verification of coding styles and different types of tests already exist, other programmers still often have to comment and change something which was not caught by the automatic detection systems.

To support programmers, the current solutions consist of for example code suggestions (GitHub Copilot [19]), bug finding (DeepCode extension [12]), or validating if the change is correct or not (deepreview [33]). However, none of the mentioned solutions are open-source and cannot be tailored for specific projects. Suiting such system for certain project is necessary, because reviews should be based not only on general code bugs, but project specific issues. Nevertheless, the mentioned solutions provide a promising problem approach and the system created for this thesis will be based on them.

Therefore, the aim of this thesis is to create an automatic system for reviewing the change in a flexible, human way. The project was done in cooperation with Nokia Corporation, which provides data for training the model and testing it. The next step will be implementing the system into a testing pipeline and using it on a day-to-day basis. This system would be able to be used in any programming project, e.g. from space industry, which would be big enough to be trained on.

In order to achieve such a goal, a neural network (NN) model suited for such solutions needs to be created. It was decided to recreate the deepreview model, which is a relatively simple model with one of the best accuracy and flexibility [33], [54]. However, since the deepreview solution was published, new technologies were researched, such as transformer layers [56] which are great for natural language processing, and different feature extraction methods [61], which can extract global, local and sequential parameters from the text. The deepreview architecture has the flexibility to implement such improvements, the model constructed in this thesis should show the same or better accuracy than the state-of-the-art system; however, that is not certain. Hence, different architectures of the created model will be evaluated and compared to choose the best one.

The thesis project was prepared in cooperation with Nokia company, which provides data in the form of code repositories with peer reviews of projects implementing 5G L1. They need to be downloaded and preprocessed. Furthermore, the type of data fed into the neural network model is very important; therefore, the presented solution uses Glove vectors [21], [46] which provides encoding of words into vectors of numbers which can be easily interpreted by a machine. Hence, the project is divided into data gathering, data processing, model construction, training and results comparison. Each of these steps was prepared and programmed by the author. Machine Learning

infrastructure was provided by Nokia Bell labs.

The rest of the thesis is organized as follows. Chapter 2 provides descriptions of theoretical topics related to thesis practicalities. Chapter 3 assess the current methods of code review. Chapter 4 describes the methodology and principle of operation of the thesis project. Chapter 5 review the experimental setup for evaluation of created systems. Chapter 6 presents the results of the constructed models and compares them to state-of-the-art. Finally, Chapter 7 summarizes the results and sheds some light on possible future work.

2 Theoretical introduction

2.1 Neural Network

A neural network is a web of neurons, can be biological or artificial [26]. Depending on the structure, each neuron might be connected to many others. Each connection between two neurons have some weight. The artificial neural networks are the mathematical representation of biological networks. They can be implemented in various ways, for example optical fibre connections [60]. However, most solutions are based on the matrix representation of the nodes and weights [4]. Recent studies [11] give high hopes for computing-in-memory methods, which can multiply matrices in analogue way, very fast, easily and in small form-factor devices [44]. The scheme of such implementation is shown in Figure 1. It intuitively shows how the matrix multiplication works and how it can work in analogue way. The signals of some amplitudes are put to the inputs and the resistors' matrix is set as the matrix of connections weights. The outputs are the voltage read on the other side of the resistors. Such solutions could be very helpful in space industry, because it allows for very low power and very fast computation of pretrained NN models. However, they are not suitable for training and constructing the AI models, but are great for their execution.

The most common implementations are still digital and done, usually on Graphics Processing Units (GPU). GPUs gives advantages over normal sequential calculations on Central Processing Units (CPU), because they allow for parallelization – processing thousands of operations at the same time [45]. They also allow for the flexibility of model architecture, because it can be easily programmed and changed, which is the crucial in research of new neural networks.

Neural networks require training to produce sensible results. Training means setting up the weights of the connections, which is usually done by minimizing the loss function of the output and ideal output. The trained network produces output which can be classification, probability, or something else, like text or image.

Artificial Neural Networks consist of different types of layers, for example: Dense, Convolutional, Rectified Linear Unit, Attention, etc. [30] Each layer provides different types of neurons connections and is known for providing different types of information. The layers which will be used in the thesis project will be described shortly in the following subsections.

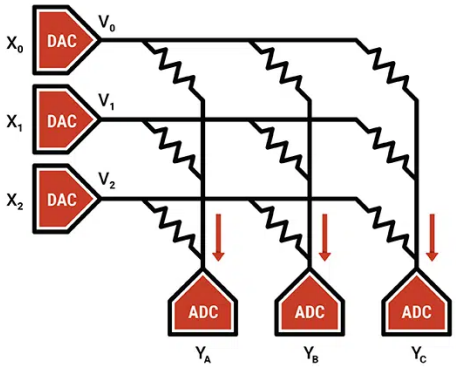


Figure 1: Illustration of variable resistors' matrix in memory chip, which can perform matrix multiplication of analogue signals. Image source: [44].

2.1.1 Dense Neural Network Layer

The Dense Neural Network Layer is the simplest NN layer, the other name is Fully-Connected NN layer. It consists of a set of neurons, where each one of them is connected to each input and each output. The output is the result of matrix, vector multiplication of the input layer and the connections weights [4], scheme of network connections shown in Figure 2. Those are universal layers, which can be used for diverse purposes, but in general they give some weights to each of the inputs, so there is no sequential info, or any neighbouring relations, the output purely depends on the input values.

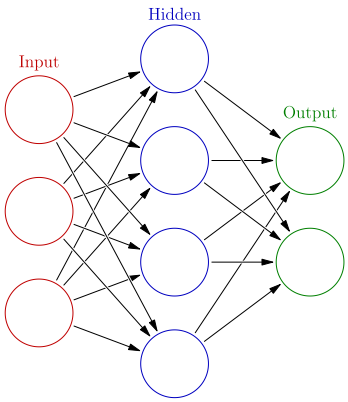


Figure 2: Scheme of simple neural network with dense hidden layer. Each of the input nodes is connected to each neuron in the hidden layer, which is then connected to each output node. Each connection has some weight which can be adjusted during NN training. Image source: [20].

2.1.2 Convolutional Neural Network Layer

The Convolutional Neural Network Layer is used to find shapes/close schemes in the input, by computing a mathematical operation called convolution.The convolution computes the overlapping of one function which is being shifted over the other.

Convolutional layer have the overlapping function defined as the vector or matrix of numbers – kernels, depending on the wanted dimensionality [29]. This kernel is being swept over each data entry (Figure 3). During training, the kernels are evolving, adjusting slightly, to give most significant values. Convolutional layers usually allow for detection of local features – how the values correlate to each other depending on their neighbours – inside the kernel.

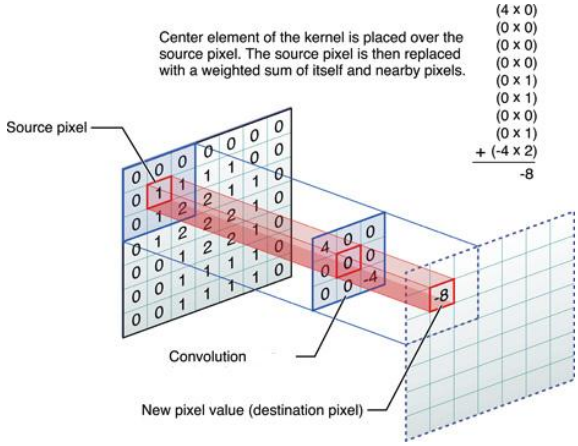


Figure 3: Demonstration of 2D matrix convolution with kernel. Each value of the matrix is being computed by calculating the weighted sum of convolution of area around the current pixel with kernel matrix of the same dimensions. Image source: [38].

In Convolutional Neural Network Layer, number of filters is being chosen, which correspond to how many kernels will be used for performing convolution.

2.1.3 Long Short-Term Memory Neural Network Layer

Long Short-Term Memory (LSTM) Layer is a type of recurrent neural network (RNN) layer, which are the looped layers. They allow to be called multiple times across the input, hence they allow for finding the periodic information. However, RNNs have a major flaw, which is long term memory. They are good at finding short periodic events, but do not cope well with the longer context. This problem is addressed with LSTM, which allows for remembering also the least often events [53].

The other type of such layer is GRU – Gated Recurrent Unit. It allows for less complexity and is recommended for use with smaller datasets. However, LSTM is still recommended for use in larger datasets.

2.1.4 Attention Neural Network Layer

Attention layer tries to replicate the natural attention mechanism, where we, as humans, tries to focus only on the most important things. There are different types of attention implementation, the first one called Bahdanau attention [1] and it successor Luong attention [34]. The second one simplified the model and allows for local or global approach. The global tries to find similarities of query in whole input

when local focus only on some context window. The Luong dot product attention is implemented and easy to use in Keras library. The dot product stands for the last operation, which is done on the outputs of the end of the attention layer model. In Bahdanau model, the outputs are concatenated.

There is a possibility to use this layer as self-attention mechanism, which tries to find the most important parts of the input. Normally the attention layer takes two inputs: query and value, but in the case of self-attention, the query and value are set to be equal.

The models which utilize attention mechanisms and recently were responsible for major advancements in NLP domain are Transformer models [6]. Their crucial part is multi-head attention layer [56]. It splits the input tensors into N parts and execute self attention mechanism on each of them. Each split should learn some other features. It allows for extracting more correlations than just usage of attention layers.

2.1.5 Max Pooling Neural Network Layer

Max pooling layer is used for extraction of only maximal values from the input, accumulate features. Each value is extracted by selecting the maximum value from the area surrounding the currently considered cell. Depending on the configuration, the selected area (kernel) can iterate over each input or jump, when configured with stride option. To mitigate the decrease in size of the output vector, the zero padding (adding zeroes around original data) is being applied to the input before processing step.

2.1.6 Activation function

The activation function transforms the output of a NN layer into values which will be fed into the next layer. They usually give output from 0 to 1, for example sigmoid, or softmax functions, but not always (relu, linear functions). Without the activation function, it is possible to solve only linear problems, because the output of the neural network is linear if none activation function is applied [14]. Comparison of two examples of activation functions – sigmoid and relu in Figure 4.

2.2 Loss function

The loss function is the way of scoring the NN, whether is coping good or badly, on the sample. It is calculating the distance between predicted label and true label, however it does not always have to be Euclidean distance. Furthermore, it usually isn't Euclidean distance, but some other function, like binary cross-entropy, or logarithmic loss, etc. The model is trained in the way to minimize the loss function. It is crucial to choose appropriate loss, because on this element depends on whether the model will be learning fast, or will get stuck in some local minimum. The most commonly used loss function for binary labels is binary cross-entropy [2]. There are others which allow for evaluating different types of output values, like categorical cross-entropy – used for multi-class classification, mean absolute error and mean squared error – for

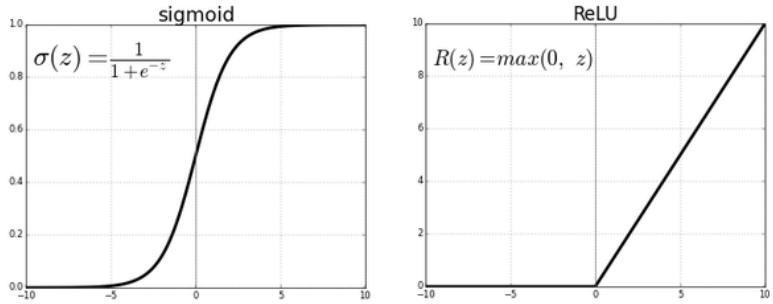


Figure 4: Comparison between sigmoid and relu activation function. The sigmoid function allows for assigning any real number to the range $[0, 1]$. It works for negative and positive values. The relu function assign 0 value to every negative input, but positive leaves unchanged. That's why sigmoid is good for classification tasks, when relu is good for filtering the values after NN layers. Image source: [8].

regression tasks and many others. Binary cross-entropy will be described in details, because it was used in the baseline model paper [33] and because it seems to be a perfect choice for the type of classification used in this thesis. The loss function for regression won't be working as well, due to their characteristics of trying to fit the values onto the function, when here the data are strictly bounded and multiple different samples have the same value of label.

2.2.1 Binary Crossentropy

The binary cross-entropy is a type of loss function used for classification of binary data – with labels $[0, 1]$ or $[-1, 1]$. The function is described with the equation [3]:

$$Loss = -\frac{1}{n}\sum_{i=1}^{n} y_i \cdot \log p_i + (1 - y_i) \cdot log(1 - p_i)$$

where:

n is output size

y_i is true label

p_i is predicted probability that the point is True

From the equation, it can be seen that the loss have two parts – for two possible labels (true and false). The function only works if the probability is from open range $p \in (0; 1)$. This type of loss function penalizes if the label is opposite to the true label. Both labels can be from range $p_i \in 0; 1$, it works also for linear regression of values in this range. Hence, it nicely suits the requirements of the thesis model and this type of loss function will be used by it.

2.3 Validation

When the model is trained, it needs to be evaluated on part of the dataset in order to see how well it performs compared to others. We are comparing many different metrics, to see what's good and what's bad in the current model (more about metrics

in Section 2.4). However, the evaluation of the model has to be independent of the training dataset. The simplest approach is selecting part of the original dataset which won't be used for validation during training and part which will be used for testing the trained model. This method is called hold out method [5]. The test dataset is used for evaluating whether the model is overfitting, because during training the model can learn too well the specifics of the validation dataset and won't be able to classify a general case. However, it is not an ideal approach, because it can happen that the selected test dataset won't describe well the general data, it can also suffer from some specific features which mostly occur in this part of dataset.

The method used to mitigate such issues is called k-fold cross-validation [5]. The implementation starts from dividing the whole dataset into k random parts. Then the model is trained using $k-1$ parts, leaving one for validation. The model is trained k times, every time a different part is used for validation purposes. At the end, the metrics are being averaged, and such values are used for evaluating the model. This method has major disadvantage – the model has to be trained k times which is very time and computational consuming. However, it allows for the best evaluation of the model, checking whether the model works well on all parts of the dataset. By analysing the standard deviation from the average, from runs using all parts it can be established how stable the model is, or how much it varies across different runs.

2.4 Metrics

Metrics are the measures used for evaluating how good the trained models are. Often one value is not enough for checking the model [35]. Let's get into details of different metrics in this subsection. The confusion matrix and analogically the metrics which are calculated from it requires predicted and true labels to be binary, when the ROC curve and its derivative metrics requires only true labels to be binary. It allows for choosing the optimal classification threshold and obtaining more detailed metrics using confusion matrix. The discussed metrics were chosen, because even through one value, cannot tell the whole story about the model effectiveness, seven of them can describe it better. They were chosen also because they are commonly used for evaluating these types of models [35].

2.4.1 Confusion matrix

Confusion matrix is the most basic way of establishing whether the model is performing good (most of the classified labels belong to TP and TN) or bad (more classified labels belong to FP and FN). It consists of four values:

	Predicted True	Predicted False
Label True	True positive (TP)	False negative (FN)
Label False	False positive (FP)	True negative (TN)

The correctly classified values are on the diagonal (TP and TN), when wrongly classified points are laid on the other diagonal (FN and FP).

True positive value is the number of points properly classified as positive. The true negative value is the number of points properly classified as negative. The false positive is the number of point which were wrongly classified as positives and false negative similarly if the points were wrongly classified as negative.

The important thing is that the confusion matrix is created for some specific threshold for establishing whether the value is negative or positive. It might be very important to set up the threshold differently than 0.5. For establishing the best threshold, read Section 2.4.7 about ROC curves and Section 2.4.8 about AUC values.

2.4.2 Accuracy

Accuracy is the value calculated from confusion matrix using equation [47]:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}.$$

It is the ratio of the number of correctly classified points to all the points. It is one of the most commonly used metric for evaluation of the model [35]. However, it doesn't say the whole truth about the classification. It can be fooled when the number of certain type of points is larger than the other. In other words, when the labels are unbalanced, because in such case the value of accuracy will be high even though the model could have classified all the points as single label. The closer the value of accuracy to 1 the better.

2.4.3 Precision

Precision is the value calculated from the confusion matrix using equation [47]:

$$Precision = \frac{TP}{TP + FP}.$$

It is in other words called confidence, as it is the ratio between correctly classified values to all positively classified values. The closer the value of precision to, 1 the more confident is that if the model will classify the point as positive, it will be truly positive.

2.4.4 Recall

Recall is the value calculated from the confusion matrix using equation [47]:

$$Recall = \frac{TP}{TP + FN}.$$

It describes the sensitivity of the model, because it is the ration of true positive labels to sum of true positive and false negative. As usual the value closer to 1 the better, furthermore, in this case the lower value, the more false negatives the model selects. It is a crucial value for establishing the ROC curve (Section 2.4.7), because it represents how sensitive the model is to the threshold change. The sum of TP and FN is constant, and the values of TP and FN are dependent on each other. Hence, changing the threshold will increase one number and decrease the other, and thus the recall measures the sensitivity.

2.4.5 Specificity

Specificity in other words, Inverse Recall or true negative rate, is the value calculated from the confusion matrix using equation [47]:

$$Specificity = \frac{TN}{TN + FP}.$$

It describes how much of true negative labels are truly negative, because it is the ratio of true negatives to how many negative points there were in the dataset. The value closer to 1 the better, However if for example the model will classify all the points as negatives, the value of specificity will be 1 even thought the model classified points wrongly.

2.4.6 F1 Score

It is one of the accuracy measures of binary classification problems which is very often used. It is calculated as the harmonic average of precision and recall. [58] As standard, they are given by equation:

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}.$$

The F1 score can be easily fooled when the model generally outputs positive output and the dataset has mostly positive labels. However, generally, F1 score of value 1 means perfect precision and recall, which don't necessary mean that the model has good accuracy. The value 0 of F1 score means that either precision or recall is 0.

2.4.7 ROC curve

ROC curve translates to receiver operating characteristic curve [17]. It is a graphical representation of binary classificator accuracy depending on the threshold which was set. It is done by plotting recall (true positive rate) against the false positive rate ($FPR = \frac{FP}{FP+TN}$). It allows for establishing the best threshold for classification. The optimal threshold is being found with method called Youden Statistics [59]. It is base on finding the max difference between true positive rate and false negative rate for each threshold. So the optimal threshold is where:

$$optimal\_treshold = threshold\,[argmax(TPR - FNR)]\,.$$

It also shows how well the model is separating both classes from each other – smooth line near the left upper corner means good separation, line on diagonal means completely random classificator. Example of the ROC curve shown in Figure 5.

2.4.8 AUC value

The AUC value translates to Area Under the Curve, and it is simply the integral of a ROC curve [17]. AUC value can be seen in Figure 5 as a greyed out area. It

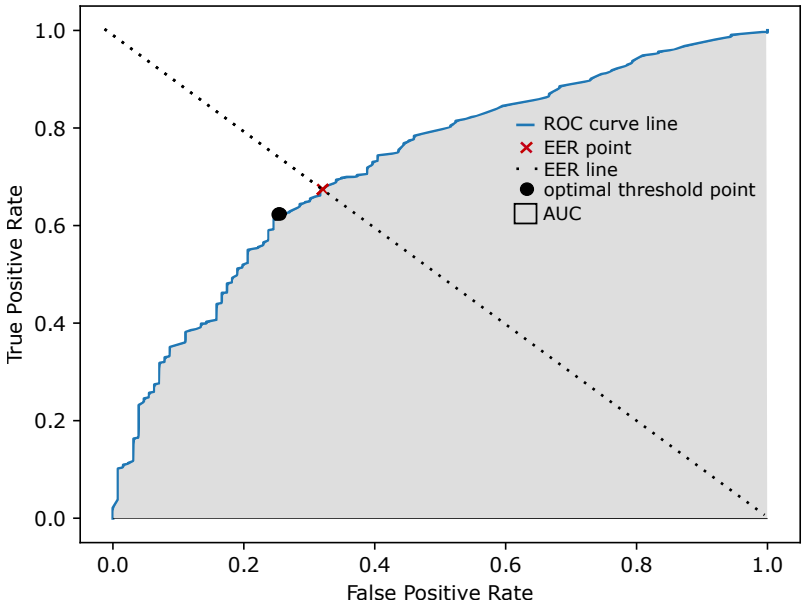


Figure 5: Example of ROC curve with explanation of AUC (Area Under the Curve) and EER. The blue line is the ROC curve line, the closer it gets to point $[0, 1]$, the better, it means that the classification is non-random. The ROC curve is closer to diagonal from $[0, 0]$ to $[1, 0]$, the more random the classifier. EER point is in the crossing of the diagonal from point $[0, 1]$ to $[1, 0]$ with the ROC curve, the closer it gets to point $[0, 1]$ the better. An optimal threshold for classification can be calculated from the ROC curve (Section 2.4.7). Area under the curve is indicated with gray coloured region under the ROC curve.

describes how well the model separates the classes in binary classification. AUC value 0.5 means the classificator is random, less than 0.5 means that something is wrong with the model, and it classifies data good but as opposite labels. Values close to 1 means that model is working well.

2.4.9 EER value

EER stands for Equal Error Rate, and it is the false positive rate in the point on the ROC curve where the false positive rate and the false negative rate are equal [55]. It can be easily read from the ROC curve, like shown in Figure 5. This is the point where diagonal $[0, 1]$ to $[1, 0]$ crosses the ROC curve. In general, lower EER means more accurate model.

2.5 Words into vector representation methods

In natural language processing, usually text have to be represented in more meaningful way than as the sequences of characters. Each word is treated as a separate unit, which is being represented as n-dimentional vector [41]. There are many methods which can achieve that, they mostly differ in the way how the words are being distributed in the artificial space. We will shortly describe two of them, which were

used before in code review problems [33], [61] and are commonly used as standard methods. Both of them are neural network based systems which have to be trained on text before use.

2.5.1 Word2Vec

It is the neural network based system for converting words into vectors and furthermore detecting synonymous words and relations patterns. It preserves the syntactic and semantic relationships, so it is possible to perform algebraically queries like, e.g.: to get the answer of relation query: "Man-Woman" for input word "brother", it is possible to compute [41]:

$$X = vector(\text{“}brother\text{”}) - vector(\text{“}Man\text{”}) + vector(\text{“}Woman\text{”}) \approx vector(\text{“}sister\text{”})$$

or to find a word that is similar to "small" in the same sense as "biggest" is similar to "big":

$$X = vector(\text{“}biggest\text{”}) - vector(\text{“}big\text{”}) + vector(\text{“}small\text{”}) \approx vector(\text{“}smallest\text{”})$$

Such results are achieved by minimizing the cosine similarities (cosine of an angle between two vectors) between the closest words in meaning. This approach favours the models which generate different dimensions of meaning.

The paper [41] describes two models for conversion of words into vectors (Continuous Bag-of-Words Model (CBOW) and Continuous Skip-gram Model (Skip-gram)) based on two previous papers [40] and [42]. The CBOW model looks at the context and predicts the word, when the Skip-gram model looks at the word and predicts the context. Both of the models use local context window for classification.

2.5.2 GloVe

GloVe translates to Global Vectors, which are used for text representation. It connects two approaches: global matrix factorization method (like latent semantic analysis [13]) and local context window [40]. The first of those is good at extracting statistical information, but deals badly with word analogy tasks. The local context window models are good for analogy tasks, but doesn't use the statistic data. Hence, this model utilizes the local context window on the global co-occurrence counts and merges the best of both worlds.

The model utilizes the relational probabilities retrieved from the co-occurrence counts matrix, which can be seen in Table 1. Using such setup, it extracts semantic and syntactic analogies and distributes the words in 300 dimensional space. It achieved higher scores than *Word2Vec* method, hence it was used in this thesis project.

Table 1: Co-occurrence probabilities for words *ice* and *steam* with 4 selected words. The probabilities are calculated by counting the co-ocurence of words in the random sequence of words from the corpus. Looking at just those probabilities we can see that both *ice* and *steam* are water, *ice* is solid and *steam* is gas, furthermore both of them are not fashion. Table cited from [33].

Probability and Ratio	k=solid	k=gas	k=water	k=fashion
P(k|ice)	$1.9 \cdot 10^{-4}$	$6.6 \cdot 10^{-5}$	$3.0 \cdot 10^{-3}$	$1.7 \cdot 10^{-5}$
P(k|steam)	$2.2 \cdot 10^{-5}$	$7.8 \cdot 10^{-4}$	$2.2 \cdot 10^{-3}$	$1.8 \cdot 10^{-5}$
P(k|ice)/P(k/steam)	8.9	$8.5 \cdot 10^{-2}$	1.36	0.96

3 Automatic code review methods

The commercial programming projects unites tens of developers working on one thing. Developers write thousands of lines of code, and they make a lot of mistakes along the way. In Nokia L1 section which author is part of are around 600 active developers, which create tens of changes every day. Some are small, others can be even dangerous for the company. Hence, it is very important to check every step of the project development. There are multiple ways to do that, but most common are: static code analysers, unit tests, environmental tests. Static code analysers are processing code without executing or compiling it. Unit tests are written by programmers to test the functionalities of each method, sometimes written before the implementation. Environmental tests are used to evaluate the function from the whole project perspective, or at least more wide than just executing a single function like in unit tests.

Automatic static code analysers compares source code against known standards and find deviations from the guidelines [22]. However, they miss a lot of problems which are program specific, which usually have to be caught by unit and environmental tests.

When the change is uploaded as a commit to the project repository, usually well configured automatic pipeline compile the change and run all the described analysers and tests to check if everything works properly and if the change is not breaking something in other places of the project.

Automatic code review methods, which uses machine learning methods focus on static code analysis, because other types of tests are very project specific and would require a lot of additional infrastructure. The first of such methods involved using TFIDF method to build vocabulary and then used Logistic Regression classifier [36] for establishing whether the change is approved or rejected. TFIDF stands for term frequency–inverse document frequency [51]. It describes how important is certain word in context of whole text corpus, assign weight to each word. Using such weights, the first methods represented the original and changed code using such system, concatenated the obtained values and used Logistic Regression classifier to predict whether there are some errors or not. Other method exploited SVM machines [57] as the classifier. Such change allowed to increase the F1 score by around 20 % [54]. SVM works in the way that it finds the best margin between data points and decision line, when Logistic Regression only finds the optimal decision line, according to classified points, but doesn't care about margin. It is sometimes slight change, but it is important, an example of such classification can be seen in Figure 6

Another approach was to use bag-of-words method to get features from the source code [31] and SVM for classification. It allowed for classification of buggy files with accuracy ranging from 43% up to 86% depending on the dataset, but in average 78%. BOW method is similar to GLOVE, used in this thesis, it generates the word representations in vectors, but doesn't try to allocate them in the word space, just give each word an index and the number of occurrences in the text.

The first model, which used Neural Network-like structure, was Deeper [58], which used Deep Belief Network based on Restricted Boltzmann Machines [25]. Those kinds

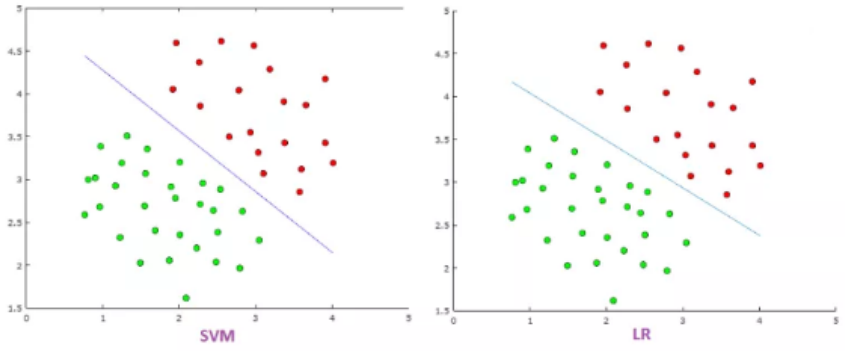


Figure 6: Classification of points using SVM classifier (left) and Logistic Regression (right). The graphs show how both classifiers make a classification line – SVM tries to maximize the margin between the line and points of both groups, when Linear regression doesn't. Image source [37].

of machines are like fully connected neural network layers, but with different algorithm for setting the weights of the connections – have undirected connections between neurons. The Deeper was tested on datasets: Bugzilla, Columba, JDT, Platform, Mozilla, PostgreSQL and got average F1 score of 0.45, when Linear Regression Classifier scored 0.25 [58].

There were also studied methods which don't analyse the code itself, but are fed with the code change properties [28], like:

- diffusion dimension - how many subsystems the change covered
- size dimension - how big the change was
- purpose dimension - what was the purpose of the change, e.g. fixing the bug, introducing new feature
- history dimension - the change history, e.g. how often the file was fixed before
- experience dimension - the developer experience

However despite the thorough research on these topics – described in many articles e.g. [24], [43], [48] and many more, which was nicely summarized in [28], the capabilities of such approach are limited, the average F1 score from the datasets the same as Deeper was 44%.

Few years later, in 2019 new state-of-the-art models occurred: deepreview and DACE. Both of them appeared in similar time, made by the same people, do not reference each other and have different values of metrics of comparing models. They also test cited models on different datasets than done on original papers. Both of the models are based on Deep Neural Networks. [33], [54]

Deep Review is taking as the input code before change, code after change and the text description. All the inputs are being encoded using word2vec technique, similar to BOW. Code is later going through a double Convolutional Neural Network

Table 2: Comparison of F1 scores of: Linear Regression model (LR), Change Parameters features (Param features), Deeper [58] and Bag-of-words models (BOW SVM). BOW SVM model was trained on a very small batch of data from the datasets – from 250 to 500 changes, so the results might be off. [31]

Project	LR	Param features	Deeper	BOW SVM
Bugzilla	0.5106	0.6147	0.6264	0.8550
Columba	0.4148	0.5550	0.5493	0.5850
JDT	0.0568	0.3616	0.3769	-
Platform	0.0603	0.3496	0.3833	0.6100
Mozilla	0.0742	0.2058	0.2213	0.5985
PostgreSQL	0.4014	0.5480	0.5463	0.4349
Average	0.2530	0.4391	0.4506	0.6150

with varying size of the filters, line by line. Text is going just through three different filter size CNNs. Then the features are fused and max pooled from all lines to get the prediction for whole change. Convolutional Neural Networks are extracting information about neighbouring words.

DACE focuses on code analysis using LSTM NN layers. They are used for extracting sequential info from the data. DACE basically combines model LSCNN described in [27] which uses CNN for extracting neighbouring words features and feeds gathered features into one LSTM layer. DACE goes one step further, adds after LSTM the PAE – Pairwise Autoencoder. Pairwise, because the autoencoder gets two inputs – old code and new code. Autoencoder is a structure which tries to reduce dimensionality of input data in the way that it can be reverted, and original data could be reconstructed. The encoder analyse the sequential info and generate from them point on the latent space, which represents the code change. It uses the reconstruction – decoding, only for training step, because the prediction whether the change is good or bad is done from the encoded data.

Comparison between simplest TFIDF models with different classifications, Deeper, LSCNN, DACE and deepreview is shown in Tables 3 and 4. LSCNN model is a model from [27], DACE model from [33], PAE is DACE model without LSTM layers, just convolutional features fed into PAE and deepreview from [33]. DACE model seems to get the highest F1 score, but there is not described what F1 score really is, and it differs significantly from the values obtained in original Deeper paper. There might have been an issue with implementation of those model because most of those models implementations are not available as open-source. Even that the DACE [54], and deepreview [33] papers are done in the same year by the same authors, they have different values of F1 score and AUC for the comparing models. The other models weren't tested on the same datasets in original papers.

Table 3: Comparison of F1 scores of: TFIDF with Linear Regression classification and SVM classification, Deeper and Deeper model with SVM classification, LSCNN, PAE, DACE and deepreview models. Data in the tables were taken from [33], [54]

Repository	TFIDF-LR	TFIDF-SVM	Deeper	Deeper -SVM
accumulo	0.227	0.239	0.202	0.199
ambari	0.240	0.278	0.306	0.238
aurora	0.204	0.220	0.349	0.299
cloudstack	0.250	0.275	0.352	0.265
drill-git	0.212	0.236	0.229	0.212
base-git	0.232	0.256	0.193	0.154
average	**0.228**	**0.251**	**0.272**	**0.228**

Repository	LSCNN	PAE	DACE	deepreview
accumulo	0.417	0.373	0.493	0.444
ambari	0.444	0.473	0.509	-
aurora	0.336	0.571	0.403	0.436
cloudstack	0.360	0.415	0.516	0.497
drill-git	0.318	0.382	0.573	0.414
base-git	0.348	0.411	0.396	0.463
average	**0.370**	**0.438**	**0.482**	**0.451**

Table 4: Comparison of AUC values of: TFIDF with Linear Regression classification and SVM classification, Deeper and Deeper model with SVM classification, LSCNN, PAE, DACE and deepreview models. Data in the tables were taken from [33], [54]

Repository	TFIDF-LR	TFIDF -SVM	Deeper	Deeper -SVM
accumulo	0.666	0.703	0.688	0.705
ambari	0.708	0.848	0.680	0.572
aurora	0.582	0.645	0.682	0.564
cloudstack	0.745	0.827	0.795	0.646
drill-git	0.658	0.725	0.593	0.540
base-git	0.679	0.759	0.590	0.524
average	**0.673**	**0.751**	**0.671**	**0.592**

Repository	LSCNN	PAE	DACE	deepreview
accumulo	0.787	0.814	0.786	0.746
ambari	0.824	0.861	0.905	-
aurora	0.750	0.819	0.793	0.758
cloudstack	0.761	0.820	0.852	0.870
drill-git	0.788	0.806	0.820	0.761
base-git	0.751	0.764	0.813	0.758
average	**0.777**	**0.814**	**0.828**	**0.779**

4 Principle of operation

In this section will be described principles of operation of each project steps. The project was divided into: data gathering, data processing, model construction and training, evaluation.

4.1 Data gathering

As mentioned in Section 1 data was gathered from Nokia gerrit. Gerrit is a code review and Git project management web application [18]. Nokia gerrit database is stored on proprietary servers, which are not easily accessible even for employees. Fortunately, there is available API which can be used to fetch the data from the database. API, is the programming interface which allows for running proper queries and searches on the database through the HTTPS protocol.

For this project sake, the scripts were created which allowed to fetch all necessary data for model training purposes. It was chosen one set of projects named *BREAM*, which consist of repositories created by L1 middleware teams from Nokia. L1 stands for layer 1 in OSI network model, which means that it is software which provides interface for higher level application to access physical medium [10]. Middleware has multiple definitions, but means a similar thing in this context, it describes that the software is responsible for connection between hardware and higher level applications [16], usually written in C/C++. Scripts were written in Python 3.6.8 [49]. For handling HTTP connection with the API, requests library was used [50]. Program was fetching list of projects inside directory "BREAM", for each project list of commits and for each commit changes in all files across all revisions, details of commit and the user comments. Such fetched data were stored in the MongoDB database, which is a simple relationless (NoSQL) database. The database structure was arranged as shown in Figure 7. The fetched data consisted of around 4000 commits, which can have multiple changes inside. The commit consists of an update to the main project, but the update can be put into the gerrit database using multiple changes to this one commit. Each change, when decided by the author that it's ready, is a subject to peer review during which developers can add flags (accepted, rejected) and/or comments. We are taking into consideration only flagged or commented changes.
So the data which was collected consist of 4144 commits with:

all flagged changes	6593
accepted changes	4272
changes with resolved comments	1008
changes with some commented issue	1257
rejected changes	56

The lines of code in collected data is approximately 4.9 million, however it is a code written mostly in C/C++, but also python and bash.
The labels of each change can be set to $0, 0.25, 0.75$ and 1:

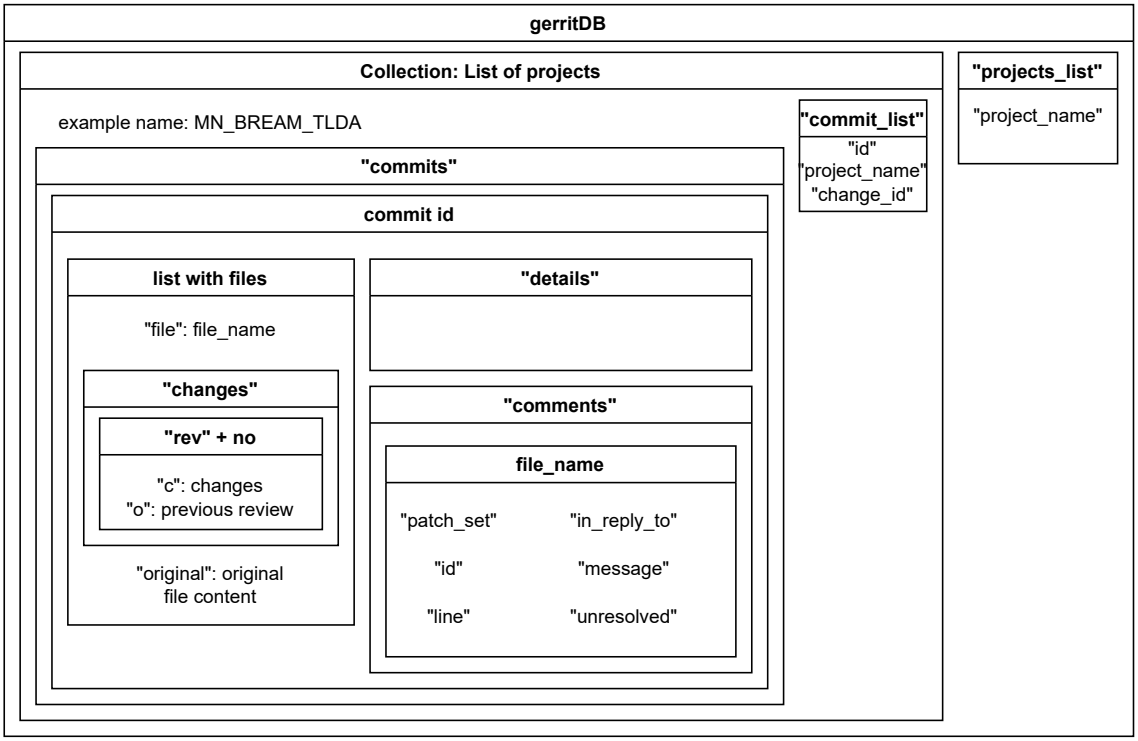


Figure 7: Layout of database used for storing code changes fetched for this thesis project. The main database consists of list of projects used during downloading the data and the repository for each project. Inside each project there is list of commits inside this project also used for downloading and the downloaded commits inside collection: "commits". Inside the commits' collection, there are commits identified using "commit id". Inside each commit is a list of changed files, details of the commit – like author, flags, etc. and comments. List of files stores the changes with all commit revision on which the changes were done for each file. The comments store all comments with the file for which the comment was made, message, etc.

Label	description
1	accepted changes
0.75	changes with resolved comments
0.25	changes with some commented issue
0	rejected changes

The labels of original changes are distributed between 0 and 1 with 4 different types of labels, but the classification task of the model remains binary, because it should allow for classification whether the change should be rejected, or is correct.

Having the labels distributed between 0 and 1 allows for approximating how probable is that the change is good or not. The intermediate labels should help to have more labels which can help model to train. However, they introduce more unknown variables to the changes labelling correctness, because not all comments can be proper and not all resolved comments means that the problem was solved. Nevertheless, without such addition, training the model would be very hard when

having 75 times more positive labels than negative [39].

The output of the model is probability that the change is good - 1 should indicate, that the change is certainly good and 0 should mean that the change is certainly bad. That's why, even though it seems that there are 4 labels, in fact it is a binary classification task. Only on the end the optimal threshold of classification is being established and the final decision made, whether the output is 0 or 1. Furthermore, when calculating some metrics like ROC curve or derived from ROC curve, the labels have to be changed into binary. In this case because of such addition of those two middle labels the results can be slightly biased and oversensitive.

4.2 Data processing

The text data cannot be easily interpreted by a binary computer, that's why it's need to be converted into more machine-readable format. There are many types of word encodings, like word2vec, GLOVE or self trained embedding layer. GLOVE was chosen, because it was proven that it has better results in word analogy representation than any other ready to use method [46]. Encoding text outside the neural network also allows for more flexibility and transparency.

In the database saved in the step before, the code of each file in change is saved as original file content and changes to it in every revision. For example:

```
"original": [
  "#include <unistd.h>",
  "#include <string.h>",
  "#include <stdio.h>",
  ...
  ],
 "changes": {
  "rev1": [
    {
      "c": [1, 70],
      "o": [1, 70]
    },
    {
      "c": [
        "        pkt->pkt.vlan_pkt.header.vlan =  (uint32_t)
            htonl(((uint32_t) VLAN_PROTOCOL_TYPE << 16) |",
        "    vlan_id);"
      ],
      "o": [70, 71]
    },
    {
      "c": [72, 143],
      "o": [71, 142]
    },
    {
      "c": [            ""          ],
```

```
            "o": [142, 142]
          },
          {
            "c": [144, 168],
            "o": [142, 166]
          }
        ],
        "rev2": [
          {
            "c": [1, 168],
            "o": [1, 168]
          }
        ]
      },
```

This is the system of saving changes designed for this thesis. As can be seen, the original file is saved as a list of strings containing lines of code. The changes contain history for each revision in which the file was changed somehow. "c" flag represents incoming change, or number of lines in current revision which corresponds to lines stored in original file, indicated by flag "o". Such architecture allows for recreating the file content for each revision.

The model has to be trained only on important parts of code, it cannot get whole files, because it would get too much information and wouldn't train at all [33]. That's why from the architecture described above are extracted only changed lines for each revision plus some margin, set to 3 lines, plus/minus. The model is getting new changed lines and corresponding old lines. The 3 line margin was chosen deliberately to give some context for the code, and also not to overweight the change itself. If the margin was smaller, some changes would have just empty lines added. It was also tested experimentally and 3 lines gave better results than 2.

Neural network is build on tensors, which requires having set lengths. Hence, the retrieved changed lines are split into hunks consisting of constant number of words. The number of words was set experimentally to 200. Such number gave fewer hunks and overall smaller data size than using 64, or 100, or 300 number of words. Furthermore, achieved the best results from tested numbers.

Extracting words from text is not as trivial task as it may seem. Natural language and code word definitions can be completely different, for example in natural language we don't need to care about capitalization, when in code this can completely change the meaning of the word, for example: *batch(* and *Batch(*. The first word is the function call or definition, where the second is object creation. Furthermore, there are a lot more words in code type of text, because the single words are often concatenation of few natural language words, like: *old_code_batch*, or *GloveEncoder*, etc. Hence, there were created two separate sets of processing criteria for natural language and code. However, they both consist of mostly adding whitespaces - which are the word separators around the following marks: "()[] {} ., "' $*$ + − | :; =". Furthermore, the end of the line symbol is also added as separate word to the dictionary, to keep lines layout.

As mentioned above, we need separate encoding for natural language text. It is so because the third input of our model is the commit message – a human-readable description of the change.

The last step of data preprocessing is encoding the words using Glove vectors. GLOVE is the GLObal VEctors representation of text. In this system there are two model types used, respectively for code and text encoding. More about Glove in Section 2.5.2. Firstly, the Glove model is trained on all the changes combined into one big file, it creates the dictionary of words, with the occurrence numbers of each word. In the next step, the Glove model is trained with following parameters, to create vectors describing words in the dictionary in relation to the whole available code:

- $vector\_size = 300$ - into how many dimensions have the output vector of each word, 300 dimensions were chosen, because according to literature that's the good dimensionality for describing code words. [61]
- $max\_iter = 100$ - maximum number of iterations, which controls the training time, by changing the number of epochs of training, set according to Glove original paper [46], for vectors with size above or equal 300.
- $window\_size = 15$ - size of context window which is used similarly like convolutional layer in neural network models, searches for local word relations, chosen 15 according to [46], [61], should be appropriate value for code text
- $x\_max = 100$ - maximum number of co-occurrences which are taken into calculations – number of times one word occurs in the context of the second word; every higher number set its weight to 1. Value set to 100 according to [46].

Generated vectors are in the form of dictionary – each word is represented by a 300 dimensional set of numbers.

The encoding is done by dividing the changed lines into hunks of constant number of words. Going into a bit of technicality, the number of individual words is approximately 100000, it makes the vector table huge, and in order to omit very slow word search in such table, the word indices in the vector table are stored separately and sorted alphabetically. Such solution allows for binary searching every corresponding word when changing the words in hunks into vectors. This optimization allows for encoding thousands of changes within few minutes.

That's how the input of the model is created – each change consists of multiple hunks – different number of hunks, each hunk consists of 200 words and each word is described with 300 float numbers.

4.3 Model construction

When the data were processed, they need to be fed into the deep learning model. The models used in this thesis had 3 inputs: text input, old code and new code. The input layers sizes were set to $(None, 200, 300)$ for code and $(None, 142, 300)$ for

text. *None* value represents the number of hunks in each change, which varies, and the layers are executed on each hunk separately, on the end only, the one value is being extracted from the outputs of each hunk. In the original model which in this thesis should be recreated, text and code are being treated a bit differently. The text feature processing block of the model is shown in Figure 8. The text input is going into three differently sized convolutional layers, followed by max pooling and flattening. Convolutional layers are computing 1D convolution comparing different numbers of neighboring words - 3, 4 and 5 and reducing the number of word vector dimensions from 300 to 70. It is not 100, like in the [33], because the larger model overflowed the GPU card memory. Max-pooling layers are configured in such way to output maximum value from 3 consecutive values of convolutional layer output. It allows for taking into account only most important connections. Flatten layer is just for being able to concatenate outputs of each convolutional branch together.

The concatenated values are fed into Dense layer, which has 100 neurons and Relu activation function, for feature detection. The last part of text processing is Dropout layer, which removes irrelevant nodes.

In code processing, there are few major differences from the block shown in Figure 8. Both code inputs goes into the same model for feature extraction – the model has shared weights between both inputs. The second difference is addition of another convolutional layer before, similar structure as text feature processing. This convolutional layer computes convolution with kernel of size 8 - it takes 8 neighbouring words and filters them using 70 different filters. This convolutional layer followed by max pooling layer, for taking only most valuable data and then dropout. Such structure output more coarse features which are layer passed through finer details filters.

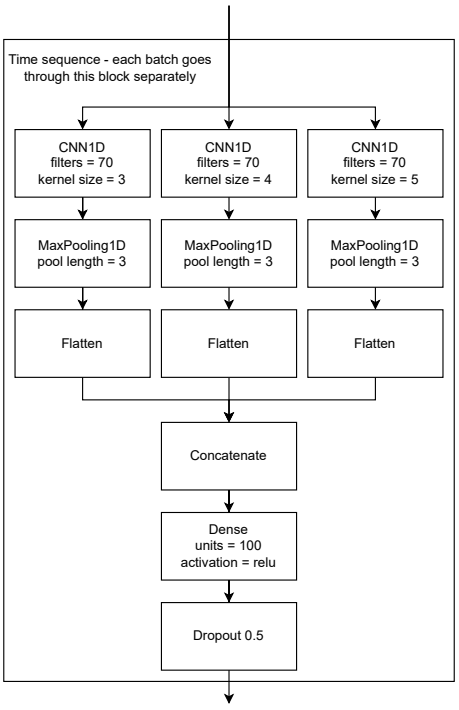


Figure 8: Feature processing block for text input of deepreview model. Whole model in Attachment A.1.

The rest of code feature extraction model is constructed like text one. After extracting the features, each of them: from text, code before change and code after change are concatenated and passed through dense layer with 100 neurons. After that there is classification layer, which is also dense layer, but with 1 neuron and sigmoid activation function. The classification is done for each hunk, so to get the classification for whole change there is one last layer, which is min pooling layer. It outputs one, minimum value from all hunks.

The feature extraction models with variable kernel sizes of convolutional layers were based on [7], [32] which provides implementation of such system [9]. This system

was also used by the state-of-the-art paper [33].

4.4 Model training

For model training, Nokia Bell Labs proprietary server is used for this purpose with Flyte management system on it. The computing cluster is made in such a way that the model is being trained on one, single GPU Nvidia A100 with 40 GB of memory. Unfortunately, due to connectivity issues between services we weren't able to monitor the training parameters with MLflow, because the models evaluated here were too big. From the same reason all the steps: model construction, training, and evaluation are executed in one Flyte task.

In the gathered dataset, assuming one hunk has 200 words, the number of hunks in the biggest change is 11785. Which means the maximum number of encoded words in one change is $7.071 \cdot 10^8$, which assuming we are using 16-bit floating point numbers and 300 dimensional Glove vectors, takes approximately 1.4 GB of space. Because the all model transformations have to be stored in the graphics card memory, it is possible to train only one hunk at once. This is the biggest bottleneck of this thesis project, because as it was found out later, achieving good accuracy with such approach is almost impossible. When training is done using a single hunk at once, minimizing the loss function is very hard, and it often gets stuck at local minima. This training approach is called stochastic gradient descent and to increase the convergence rate, the use of momentum was researched [52], but after short testing was decided not to use it, to reduce the amount of variables which have to be tested and optimized.

4.5 Model evaluation

The models were firstly evaluated using 10% of all available changes, which were not part of the training. However, because of instabilities in results, which greatly differed between the runs (e.g. AUC value in one run 0.717 and in the other 0.644) k-fold validation was used. In this case, it was chosen 5 folds, so the dataset was divided into 5 equally sized parts. In each iteration 4 of the folds were used for training and 1 for validation in addition to previous 10%. At the end, the results were averaged, and final metrics were achieved.

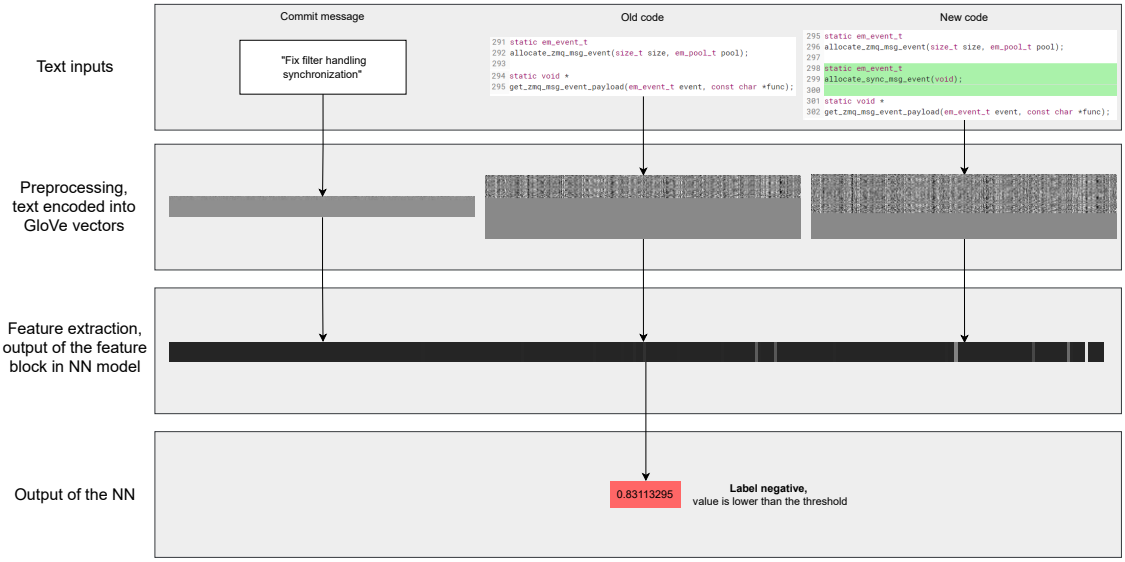
The predictions are made for each change and compared to true label. There are being computed ROC curve and AUC value. For the best found threshold the confusion matrix is created, and the following metrics are computed: F1 score, accuracy, recall, specificity, and precision. During this stage, all the plots are generated and saved.

4.6 Data going through the model

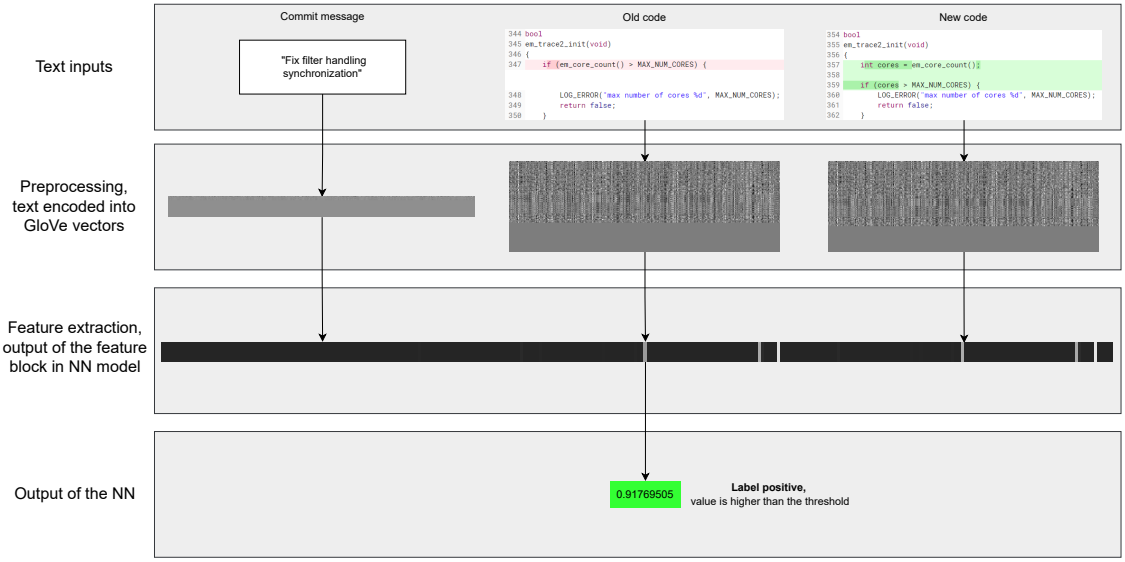
Summarizing the principle of operation of the thesis, Figures 9a and 9b show schemes of single hunk flowing through the different stages of the project. From the top are text inputs from data gathering part: commit message, old code and new code –

changed. Later there is preprocessing stage, where the text is converted into separate words and at the end into a list of vectors. The images show encoded text into GloVe vectors. The different sizes are just from trimmings of the irrelevant data, because each hunk has a constant number of words, which in most cases are just zero values. Below preprocessing is output of the feature block of the constructed model. It is a vector of size 300 which consist of values extracted from the inputs – in the left part there is 100 values extracted from commit message, in the centre there is 100 values extracted from the old code and on the right 100 values from new code. There can be seen that the old code features and new code features are a bit similar, but the new code has higher values. The bottom part is the decision, where from 300-long vector only one number is obtained. In Figure 9a the decision is negative, because the value is lower than the threshold, which in this case was equal to 0.9030617. The threshold is learned from the validation dataset. Figure 9b shows the hunk which outputs positive label.

In terms of determining whether the whole change should be rejected or accepted, the decision is done by selecting the minimum decision value from all the hunks. The below figures show examples from deepreview model constructed as described in Section 4.3 and evaluated more deeply in Section 6.1. Similar schemes were also prepared for a model with more advanced feature extraction block, described in Section 5.6 and evaluated in Section 6.5. Such schemes are in Attachments A.6 and A.7



(a) Data flow through the deepreview model and achieve negative label.



(b) Data flow through the deepreview model and achieve negative label.

Figure 9: Scheme of single hunk flowing through the deepreview model and achieve negative label in Figure 9a and positive label in Figure 9b. On each scheme there are four rows which describe different stages of data. From the top are the inputs – commit message, old code before change and new code after change. Below, the inputs encoded using glove vectors are shown. They have different sizes only because they were trimmed from some trailing zeroes, normally they all have 200 glove vectors of size 300. Below, the vector is shown which is an output of feature extraction of deepreview NN model. It has three parts of the same size corresponding to each encoded glove vectors. Based on such extracted features, the single value is calculated – last row, the output of the NN. Later, this value is used to classify whether the change is good or bad using some threshold.

5 Experimental setup

The experimental setup consist of two separate parts:

- Data gathering, preprocessing, batching
- Model construction, training and evaluation

The parts are executed on two different servers. It is done in such manner due to privileges restrictions on the high computational capabilities' server – unable to create custom Python virtual environment and install required packages.

The data are being fetched, processed and saved in *.mat* files, not decoded into global vectors, but compressed, stored just indices to vector map (more in Sections 4.2 and 4.1). Steps executed on the processing server:

1. Gathering data from gerrit
2. Construction of text for GloVe training
3. GloVe model training
4. Encoding the content of changes into GloVe indices
5. Sending the data to second server

For experiment purposes, if data wants to be updated for training, all the steps needs to be repeated. However, if for example just the bunch size needs to be adjusted, just two last steps will be sufficient. Each step is a separate script which can be configured and executed.

On the server equipped with GPUs, the Flyte workflow is used to construct a model, train it and evaluate. The principle of operation of this part is described in Sections 4.3, 4.4 and 4.5. The experiment is performed as follows:

1. change configuration of the model
2. build the container for performing actions on the GPU
3. execute the build workflow
4. download the results and analyse

After baseline of a simple working model was achieved, the following experiments were conducted:

1. reproduce baseline model [33]
2. train the deepreview model on the same dataset multiple times
3. uniform the label distribution, by weighing the loss function
4. add Attention layers - Attachment A.2

5. add LSTM layer - Attachment A.3
6. add Multi-head Attention layer - Attachment A.4
7. implement feature extraction similar to [61]

Which results in constructing model architectures (Attachments: A.1, A.2, A.3, A.4) The reproduction of state-of-the-art paper [33] is described in Section 4.3.

5.1 Experiment: Training on the same dataset multiple times

The first experiment is to train the prepared deepreview model on the same dataset multiple times. It is done to show the instabilities of the training. During the work on the model, it was observed that the metrics across different training executions are varying a lot. The cause of such behavior is the Stochastic training – change by change. Such training manner is prone to getting stuck in some local minima which are hard to escape. That's why this experiment was performed to show how much the metric vary between each training. It is using k-fold mechanism for evaluation of the results, but instead of applying different folds, it is putting in every iteration the same fold of dataset.

5.2 Experiment: Data distribution uniforming

Another experiment doesn't add any layers to the model, but was to uniform the label distribution. Because as it was said in Section 4.1 there are approximately 4 times more positive labels than negative ones. It is a problem, because if the model will always output positive labels, in theory it can have even 80% false accuracy. To prevent that, the labels during training were weighted appropriately to the labels' distribution. However, because we use binary classification, and we want to have linear value, what is the probability that the change is good, the input labels can be only 0 or 1 if we want weights. It is an issue, because in normal case we are feeding the model with 4 labels - $0, 0.25, 0.75, 1$. Labels need to be rounded to the integer value, and that might affect the final accuracy after weighting.

5.3 Experiment: Attention layers

The attention layers should increase the weights of the important nodes. If the data are too fluctuant then it won't give good results, but sometimes it can increase the model accuracy significantly. That's why, as can be seen in Attachment A.2 the attention layers were inserted after the feature extraction phase. It should help select only the most important features. Adding too many attention layers decreased the accuracy, so only those three layers were finally settled to be the best.

5.4 Experiment: LSTM layer

The LSTM layer which allows for analysing sequential data was added after merging the outputs of convolutional layers, like it can be seen in Attachment A.3. It was configured that LSTM layer outputs one vector for all the hunks, which is later just classified using single Dense layer with sigmoid activation function. As can be seen in Attachment A.3 the LSTM layer was also put into bidirectional block, which means that it evaluate the sequences in both ways.

It was decided to use LSTM layer with 100 units, to output the same size as the input – Dense layer. There wouldn't be any point in adding more units, because it would increase the dimensionality instead of reducing it.

5.5 Experiment: Multi-head attention layer

The Multi-head attention layer which is the core of the Transformer network allows for finding correlations between different parts of input. By adding this type of layer after the feature extraction part (as can be seen in Attachment A.4) it should increase the accuracy of the model, because it should find correlations between different features and pay more attention to them.

It was decided to use MultiHeadAttention layer from Keras module with 4 attention heads each having 20 dimensions. It allowed for compilation without memory overflow and significant dimensionality reduction. It is also significant that only a few points in the code usually makes hunk unacceptable.

5.6 Experiment: Advanced feature extraction approach

After implementing feature extraction from [61] it appeared that unfortunately with such dataset and such data feeding approach the model is too big to be able to train. However, after huge sizes reductions, it was possible to create a working and trainable model. It probably could be larger when not taking into account the biggest changes, but it wasn't tried.

The principle of operation of this method is different from the rest, because it is based on extraction of three types of features: global, local and sequential. The feature processing block

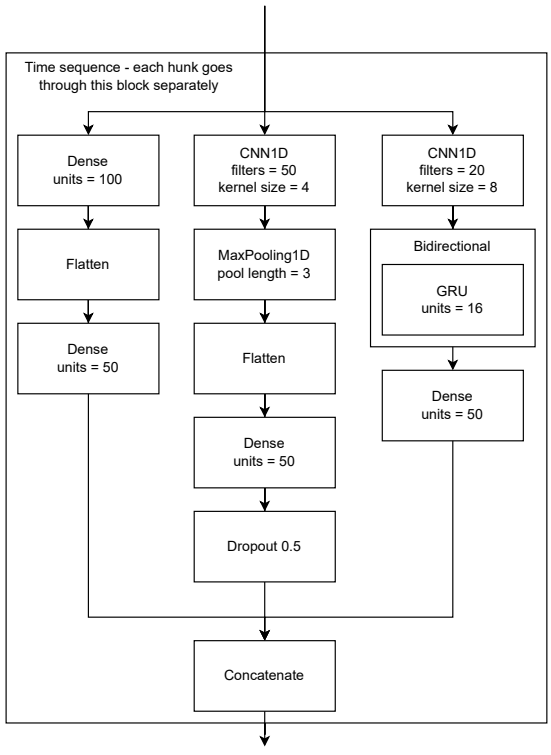


Figure 10: Feature processing block for text input of 3 types of features model. For code inputs, the block is almost the same. Whole model in Attachment A.5.

for text input is shown in Figure 10. Global features (first from the left in Figure) are extracted using fully connected layers, they take flatten input and connect it with 100 neurons. Local (middle branch) is based on the same principle as deepreview - it has convolutional layer for local context extraction. However, due to memory limitations it has only one convolutional layer with 50 filters and single kernel size of 4, which takes only 4 neighboring words into account. The sequential features (right branch) firstly extract local context information using convolutional layer with 20 filters and larger kernel size 8 to take more information into account. This first step is done purely to reduce the size of input data going into bidirectional GRU layer with only 16 units. The output sizes of branches were also reduced down to 50 units each. The features like in previously described models are being concatenated and classification is done similarly. The whole model can be seen in Attachment A.5.

6 Results

The system for gathering the code changes with peer reviews from Nokia gerrit was created and data were successfully gathered and processed using Glove [46] method. The words' separation system was researched and written. The baseline deepreview ML model [33] was recreated from scratch, implemented and the experiments were conducted (Section 5). With hardware as our limiting factor, the GPU card memory, the models were trained one batch at the time, which greatly reduces the reliability of training. It was observed that increasing the number of epochs above 3 didn't change the training loss and increased chance of overfitting (Figure 11), so most of the experiments were done using 3 epochs. When the model was trained for more epochs, it increased the chance of the model getting stuck outputting a single value for all the changes. As can be seen in the Figure 11 the training loss is higher than validation. It is caused by non-trainable layers, like dropout, attention, regularization, which offsets the loss during training, but are not taken into consideration in validation stage.

For the experiments, the hunk size was set to 200 words. All the experiments were validated using k-fold validation method. In addition to test fold in each iteration, there was separated 10% of whole dataset which was used only for validation purposes.

The results presented below are the weighed average measurements from all folds and validation split. The ROC curves show stacked curves from all folds, which can tell about model stability. In the models where it was possible to train them longer than 3 epochs also the training loss plots are attached. The confusion matrices are the sum of confusion matrices from all folds. Each metric is given with standard deviation value to visualize also the stability of training across different folds.

6.1 Deepreview

Deepreview validated using k-folds

The first step was to recreate the deepreview model. The scheme is shown in Attachment A.1. The filter size of convolutional layers was reduced to 70 from 100 used in original paper, to omit the memory overflow. The ROC curves from five different folds is shown in Figure 12. As can be seen in Figure 13 the model was trained for 10 epochs, because it was observed that the test loss was getting slightly smaller till this moment. This figure shows training loss evolution on one of the folds.

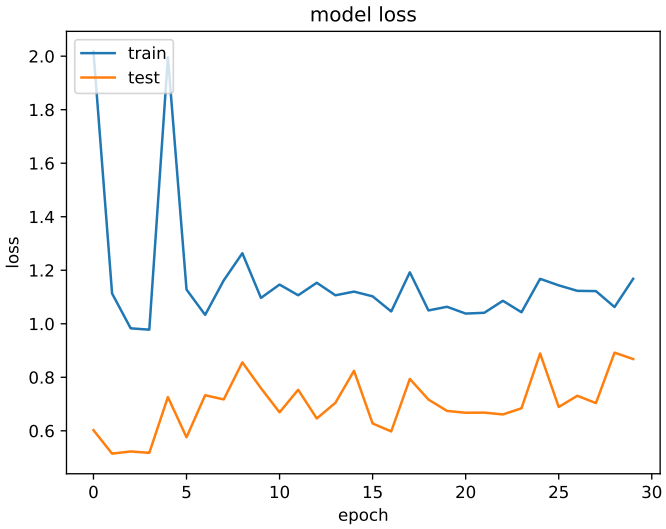


Figure 11: Training loss evolution through 30 epochs of training deepreview model. As can be seen, after epoch 3 the loss fluctuate, but doesn't decrease, that's why the epochs for most experiments were set to 3. Furthermore, the training loss is higher than the validation. It is caused by non-trainable layers, like dropout, attention, regularization, which offsets the loss during training, but are not taken into consideration in validation stage.

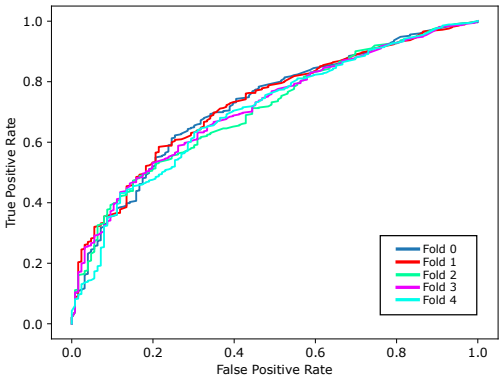


Figure 12: Deepreview model ROC curve of different folds.

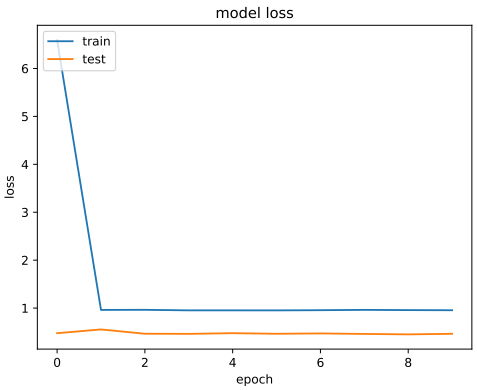


Figure 13: Deepreview model training loss across 10 epochs, from one fold.

The confusion matrix from all runs:

	Predicted True	Predicted False
Label True	4169	3091
Label False	403	1405

The metrics values are:

AUC	0.715 σ0.021
EER	0.344 σ0.022
accuracy	0.615 σ0.036
precision	0.912 σ0.011
recall	0.574 σ0.057
specificity	0.315 σ0.069
F1	0.703 σ0.043

It can be seen from the confusion matrix that the numbers of predicted ones and zeros are almost equal. It might indicate highly random prediction – the model might give labels true or false randomly – with 50% accuracy. However, it can be seen (in Figure 12) that the ROC curve is non-random and the classification is made with some confidence, however it is not a highly accurate classification.

Deepreview trained multiple times on the same data

In order to check the stability of the training, the deepreview model was trained on the same data 5 times and the variability of the metrics was observed.

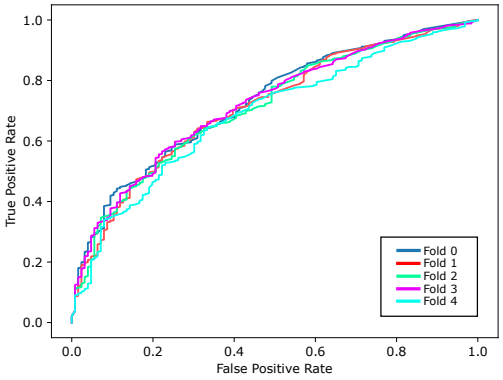


Figure 14: Deepreview model ROC curves of different trainings of the same data.

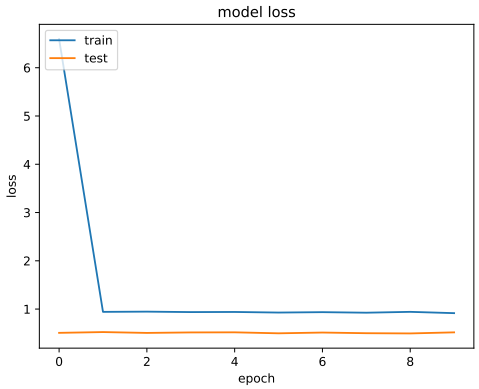


Figure 15: Deepreview model training loss across 10 epochs.

The confusion matrix from all runs:

	Predicted True	Predicted False
Label True	3822	3288
Label False	439	1521

The metrics values are:

AUC	0.699 σ0.013
EER	0.363 σ0.013
accuracy	0.589 σ0.034
precision	0.897 σ0.007
recall	0.536 σ0.060
specificity	0.369 σ0.089
F1	0.669 σ0.046

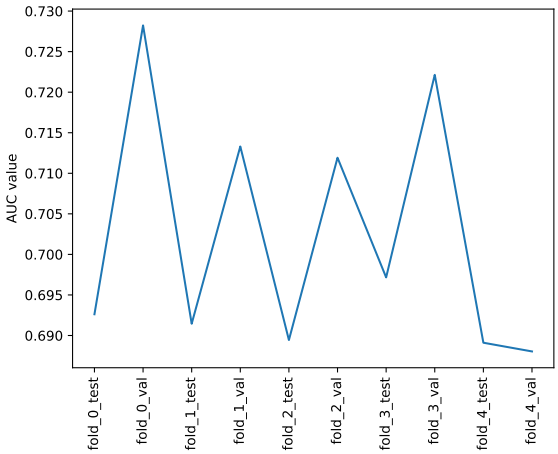


Figure 16: AUC value fluctuations across 5 folds. There are 10 points, because for every fold 10% validation part and test datasets were evaluated separately and combined only for final average metrics.

In confusion matrix, it can be seen that there are 13% more predicted negative values than positives. It might be caused by the last layers which is taking minimum value of hunk predictions. Such operation might sometimes offset the classification to negative prediction, but the negative classification is less often and more meaningful. The Figure 16 shows how the value of AUC oscillates across consecutive trainings of the model. The model was trained on the same data each time, so in a perfect case, the AUC value should be the same, but it highly oscillates. In this plot, the *fold* means each training and the *test* it test dataset, when *val* means validation data. Generally the validation dataset has a bit higher AUC values, but it also is not a rule, because in the last training the AUC value dropped to 0.688. The max value of AUC in this example was 0.728. The standard deviations of the metrics are a bit lower than when the dataset was trained in k-fold validation manner, but still high enough to show the problem of the instabilities. As can be seen in the Section 6.6 the AUC values of the model trained multiple times on the same data has minimal and maximal values smaller and bigger than the models highly modified in many different ways. That can suggest that we can only see the trend in the model values, but they won't be precise, even averaged in k-fold validation, because of such variability in training.

The other thing can be seen in Figure 14, the last fold – Fold 4, which is just 5th training on the same data, highly differs from the rest. It can suggest that in this run the model fell into some local minima. The loss function – Figure 15 shows similar learning trend in Figure 13, which is reasonable, because it is just the loss of training on one of the folds.

Deepreview trained with labels weighting

In order to address the issue of unbalanced dataset – a lot more positive labels than negative, hence the labels weighting was applied as described in Section 5.2.

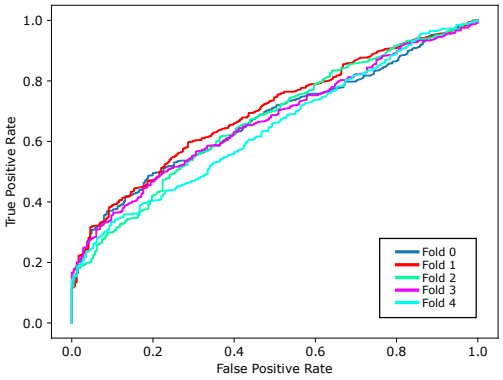


Figure 17: Deepreview model ROC curves with weighted labels from 5 fold validation.

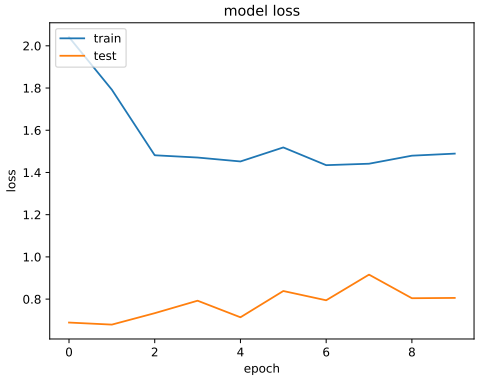


Figure 18: Deepreview model training loss across 10 epochs.

The confusion matrix from all runs:

	Predicted True	Predicted False
Label True	3406	3854
Label False	356	1452

The metrics values are:

AUC	0.666 σ0.025
EER	0.391 σ0.025
accuracy	0.536 σ0.061
precision	0.909 σ0.021
recall	0.469 σ0.092
specificity	0.411 σ0.123
F1	0.613 σ0.077

As can be seen in Figure 18 the training looks a lot different from the deepreview model without uniform labels distribution (Figure 13). There can be seen signs of overfitting in the increasing test loss. Furthermore, the ROC curves (Figure 17) are more spread than in the original model, and they represent a lot more random label distribution. The high trend for classifying negative labels- 5306 negative and 3762 positive labels, suggests the minimum pooling problem. The last layer is taking minimum value of hunk prediction, which makes the model more prone to detecting faulty changes – because the faulty change is indicated as lower label and there are less negative labels in the dataset. The metrics of accuracy, AUC, F1 score are very low compared to the original deepreview model.

6.2 Deepreview with attention layers

This model differs from the deepreview recreation by addition of attention layers. Its scheme is shown in Attachment A.2. When this model was trained with 10 epochs, like deepreview, it overfitted and got stuck to a single output value, hence 3 epochs were used and that's why there is no loss plot, because it has just 3 points. The ROC curves from all folds are shown in Figure 19

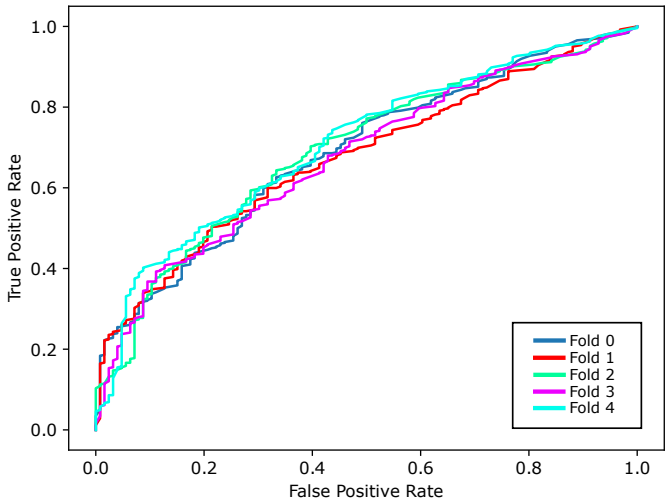


Figure 19: Deepreview model with attention ROC curves from 5 fold validation.

The confusion matrix from all runs:

	Predicted True	Predicted False
Label True	4013	3247
Label False	450	1358

The metrics values of the best run are:

AUC	0.691 σ0.014
EER	0.357 σ0.013
accuracy	0.592 σ0.045
precision	0.901 σ0.022
recall	0.553 σ0.073
specificity	0.317 σ0.089
F1	0.682 σ0.056

It can be seen in Figure 19 that the ROC curves are a bit more spread, it means, that there are even more instabilities in training compared to pure deepreview model. The ROC curves seem to be not random, however this modification decreased most of the metrics, which means that it wasn't a good change. It probably highly overfitted even just after 3 epochs, because the output values were very close to each other.

6.3 Deepreview with LSTM layer after feature processing block

This model differs from the deepreview recreation by addition of LSTM layer. Its scheme is shown in Attachment A.3. The ROC curves from all folds are shown in Figure 20. The training was done using 3 epochs, because of later overfitting and the loss function is not shown, because it has only 3 points.

The confusion matrix from all runs:

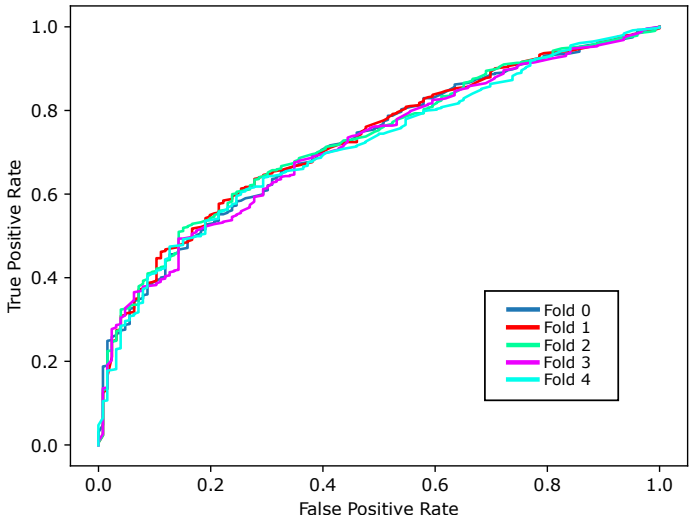


Figure 20: Deepreview model with LSTM ROC curves from 5 fold validation.

	Predicted True	Predicted False
Label True	4158	3102
Label False	396	1412

The metrics values of the best run are:

AUC	0.723 σ0.019
EER	0.339 σ0.019
accuracy	0.614 σ0.037
precision	0.913 σ0.014
recall	0.572 σ0.056
specificity	0.316 σ0.061
F1	0.702 σ0.045

This is the model which has the best metrics from the models derived from deepreview. The ROC curves shown in Figure 20 are very consistent, and also their shape is most promising, like the baseline model, Figure 12. The model has higher value of AUC and precision and lower value of EER compare to baseline. However, the number of predicted positive values is only 40 more than negative values, which is equal to 0.8% of difference. It might suggest that the model is only guessing and have 50% chance of giving positive or negative flag. Furthermore, such behaviour could benefit from the non-uniform distribution of labels. However, the ROC curves show that the distribution of the points wasn't random, and adjusting the threshold allowed for good classification.

6.4 Deepreview with Multi-Head attention layer after feature processing block

This model differs from the deepreview recreation by addition of Multi-head attention layer. Its scheme is shown in Attachment A.4. The ROC curves from all folds are

shown in Figure 21. The training was done using 3 epochs, because of later overfitting and the loss function is not shown, because it has only 3 points.

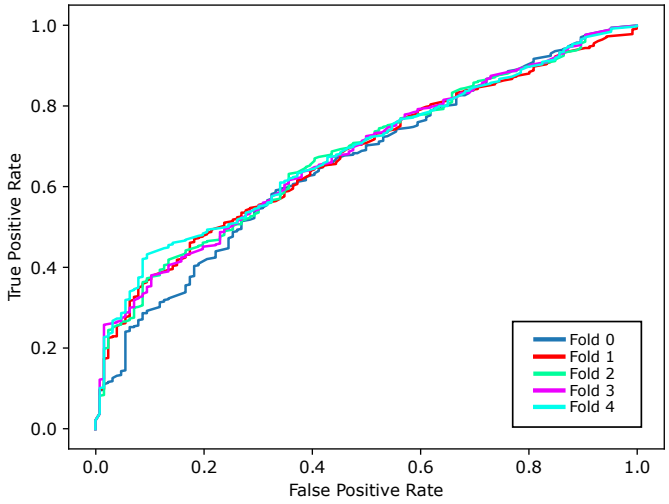


Figure 21: Deepreview model with Multi-Head attention layer ROC curves from 5 fold validation.

The confusion matrix from all runs:

	Predicted True	Predicted False
Label True	3770	3490
Label False	371	1437

The metrics values of the best run are:

AUC	0.700 σ0.032
EER	0.361 σ0.027
accuracy	0.574 σ0.057
precision	0.914 σ0.030
recall	0.520 σ0.088
specificity	0.365 σ0.099
F1	0.657 σ0.068

The ROC curves - Figure 21 seem to be highly random and unstable. The model doesn't show improvement from the base-model. However, it has the highest value of specificity from all models, however that doesn't help the flaws of this model.

6.5 Advanced feature extraction model

This model, created based on [61], is described in Section 5.6 and its scheme is shown in Attachment A.5. The ROC curves from all folds are shown in Figure 21. The training was done using 3 epochs, because the model overfitted very quickly.

The confusion matrix from all runs:

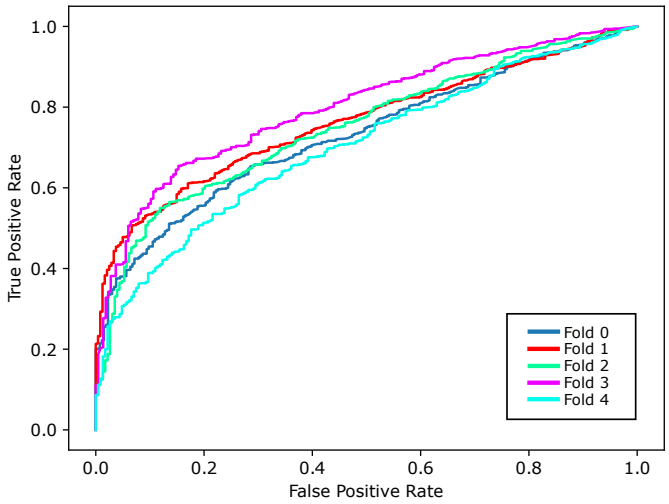


Figure 22: Advanced feature extraction model ROC curves from 5 fold validation.

	Predicted True	Predicted False
Label True	4338	2922
Label False	372	1436

The metrics values of the best run are:

AUC	**0.736 σ0.030**
EER	**0.327 σ0.019**
accuracy	**0.637 σ0.041**
precision	**0.922 σ0.021**
recall	**0.597 σ0.065**
specificity	0.315 σ0.080
F1	**0.722 σ0.047**

The ROC curves in Figure 22 seem to be unstable – they are spread between runs, but have rounded shape and are closer to the point $[0, 1]$ than for the previous models. This model wins in almost all metrics compared to recreated deepreview and its derivatives. However, it has only 8% difference between predicted positive and negative flags, which seems it's not random, although its metrics are higher due to the non-uniform distribution of labels. It is fluctuating a lot from one training to another, but the AUC value in those 5 folds reached as high as 0.798 and F1 score 0.775, which are very promising values for future improvements.

6.6 Comparison between experimented models

Using k-fold validation, the experimented methods were compared. Number of folds was set to 5, because this number allows for good amount of training/testing data ratio - 80% to 20%. Any fold more increases the training time significantly, because the model has to be trained for every fold. In Table 5 are shown the results of comparison of prepared models. The weighting of labels is not shown here since

it was not a modification to the model, and it didn't show any improvement to the baseline deepreview model. As can be seen, the model which is not based on deepreview architecture, but utilizes more complex feature extraction outperforms other models.

Table 5: Comparison between models: deepreview A.1, Attention A.2, LSTM A.3, Multi-Head attention A.4 and Advanced Features A.5. Values in gray background and bold indicates the best model value for each parameter.

Metrics	deepreview	Attention	LSTM	Multi-Head att	Advanced feat
AUC	0.715 σ0.021	0.691 σ0.014	0.723 σ0.019	0.700 σ0.032	**0.736 σ0.030**
EER	0.344 σ0.022	0.357 σ0.013	0.339 σ0.019	0.361 σ0.027	**0.327 σ0.019**
accuracy	0.615 σ0.036	0.592 σ0.045	0.614 σ0.037	0.574 σ0.057	**0.637 σ0.041**
precision	0.912 σ0.011	0.901 σ0.022	0.913 σ0.014	0.914 σ0.030	**0.922 σ0.021**
recall	0.574 σ0.057	0.553 σ0.073	0.572 σ0.056	0.520 σ0.088	**0.597 σ0.065**
specificity	0.315 σ0.069	0.317 σ0.089	0.316 σ0.061	**0.365 σ0.099**	0.315 σ0.080
F1	0.703 σ0.043	0.682 σ0.056	0.702 σ0.045	0.657 σ0.068	**0.722 σ0.047**

6.7 Comparison with baseline method

The baseline method - deepreview [33] had an average AUC value of 0.779 and F1 score of 0.451. The F1 score wasn't defined in [33], but we might assume that this F1 score is in fact half of the F1 score according to definition (Section 2.4.6), because of the values the authors are claiming for other models. If that is the case, the true F1 score of deepreview model tested on different datasets is 0.902.

Models recreated in this thesis doesn't get close to those values, at best they achieved $AUC = 0.736$ and $F1 = 0.722$ with advanced feature extraction model (Section 5.6). Such modification to deepreview seemed to give the best results. Unfortunately, we weren't able to get an implementation of deepreview model from its authors and so, there might be some differences which makes it less effective. However, the most probable cause of such results is difference in dataset. The dataset in this thesis wasn't filtered in terms of only relevant changes, or comments to changes. The thesis also used experimentation half-negative and half-positive labels, using code comments, which wasn't used before. The model sizes had to be reduced drastically, and it had affected the accuracy.

7 Summary

The aim of the thesis project was to create and study a machine learning model for automatic code review of 5G L1 Nokia projects. As can be read in Section 6 it was successfully achieved. The Deep Review NN model [33] was recreated and experiments to improve its accuracy were performed.

The studied models are getting commit messages and code changes from the commits. The models with varied or expanded input information could be investigated. Some commit information like comment message was left unused. The model which use other information than just from the neighbouring words, like deepreview, seem to have better effectiveness in classifying data. Even after great reduction in layers sizes, it had better metrics values than the baseline model. That's why the most of the work should be put into preprocessing step and changing the structure of fed data. Preprocessing described in this thesis was done simpler, because this system is complex and there wasn't enough time to refactor and recreate it after implementation.

However, the accuracy and AUC values levels similar to described in original papers [33] were not achieved. There are multiple reasons for such failure. The most probable one is that the dataset wasn't preprocessed enough. In Nokia gerrit repository, the vast majority of changes are at some point finished and merged successfully. In the whole set of projects, only few commits were rejected. Using such data, it is not possible to train well neural network. To avoid this issue, the approach of setting the flag to 0.25 (a bit better than full rejection flag of value 0) of commits with some comments was used. It increased the number of rejected changes significantly. However, this approach has major flaw, some comments are wrong or just informative. Only a quick keyword search was used to avoid this issue, but unfortunately with such big number of changes it wasn't checked thoroughly.

Another issue which can cause difference in accuracy might be the difference in data which are fed into the model. In the thesis project it was given whole changed code, with some margins, without indicated lines. It is not clear how the changes looked in the deepreview paper, but possibly the changes were smaller and only the relevant lines were given to the model. Although, it would be hard step, because final model have to use whole change to establish whether it is good or bad and shouldn't rely on only picked lines by developer. This could be changed in such a manner that only the few picked hunks from each change would be used for training, but the prediction would be done on the whole change. That would allow for increasing the size of advanced feature extraction model and possibly better accuracy. Because even after such reduction of sizes and major architectural changes, this model outperformed the recreated baseline model.

Even the baseline model and its derivatives layers sizes had to be reduced. The major difference was the convolutional layers filters sizes reduction from 100 to 70. That was a sane decision to allow training at all due to memory limitations. The size of the data would have to be reduced significantly to allow fluent training with such big filter sizes.

7.1 View for the future

Emphasis should be put on the preprocessing stage, providing only relevant information to the model. That would allow for reducing memory needed for model training, hence for training more hunks at the same time, giving more accurate gradient descent of the loss function. That should solve, or decrease, the loss function oscillation problem. It should be also experimented with the model trained on the cherry-picked lines, which are wrong, or solved, and in the prediction stage, the whole change should be applied, and the minimum prediction should be chosen.

The important step would be also to analyse with better working models how the input data affects the prediction. The experimental data analysis should be performed in order to know what parts of model should be changed and what kind of preferences the model has. It was performed in very limited way because of the system architecture, on which the models were trained, and because obtained models results were too random. That kind of data analysis could look for correlations between some input parameters and how they affect model training and output, for example, whether some words are more prone to fail the whole change, etc.

As can be seen in [54] paper, even better results can be obtained using convolutional autoencoders, which were very popular for sound generation few years ago [15] and nowadays are being replaced with Variational Autoencoders [23]. They allow for more continuous distribution of points in latent space and could allow for better prediction of data which differ in significant way from trained data.

If the proposed experiments would allow for creation of a code review model which would have sufficient accuracy, it might be tried implementing it into the automatic testing pipeline. Such systems are a promising solution for reducing the amount of time the programmers need to spend on manual code review of other developers.

References

[1] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[2] A. Bhardwaj, W. Di, and J. Wei, *Deep Learning Essentials: Your hands-on guide to the fundamentals of deep learning and neural network modeling*. Packt Publishing Ltd, 2018.

[3] *Binary Cross Entropy Loss function - documentation of PyTorch 1.13 library*, https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html#torch.nn.BCELoss, Accessed: 2022-07-15.

[4] C. M. Bishop *et al.*, *Neural networks for pattern recognition*. Oxford university press, 1995, ch. 1.

[5] C. M. Bishop *et al.*, *Neural networks for pattern recognition*. Oxford university press, 1995, ch. 9.8.1.

[6] T. Brown, B. Mann, N. Ryder, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[7] Y. Chen, "Convolutional neural network for sentence classification," M.S. thesis, University of Waterloo, 2015.

[8] *Comparison of Sigmoid and Relu functions*, https://www.quora.com/Does-Softmax-work-better-than-ReLU-as-the-output-layer-in-CNN-neural-networks, Accessed: 2022-06-15.

[9] *Convolutional Neural Network For Sentence Classification - implementation*, https://gist.github.com/shagunsodhani/9ae6d2364c278c97b1b2f4ec53255c56, Accessed: 2022-06-15.

[10] J. D. Day and H. Zimmermann, "The osi reference model," *Proceedings of the IEEE*, vol. 71, no. 12, pp. 1334–1340, 1983.

[11] M. Dazzi, A. Sebastian, L. Benini, and E. Eleftheriou, "Accelerating inference of convolutional neural networks using in-memory computing," *Frontiers in Computational Neuroscience*, p. 63, 2021.

[12] *DeepCode VScode extension*, https://github.com/DeepCodeAI/vscode-extension, Accessed: 2022-02-20.

[13] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, pp. 391–407, 1990.

[14] B. Ding, H. Qian, and J. Zhou, "Activation functions and their characteristics in deep neural networks," *2018 Chinese control and decision conference (CCDC)*, pp. 1836–1841, 2018.

[15] G. Elhami and R. M. Weber, "Audio feature extraction with convolutional neural autoencoders with application to voice conversion," 2019. [Online]. Available: http://infoscience.epfl.ch/record/261268.

[16] L. H. Etzkorn, *Introduction to Middleware: Web Services, Object Components, and Cloud Computing*. Chapman and Hall/CRC, 2017, ch. 1.1.

[17] T. Fawcett, "An introduction to roc analysis," *Pattern recognition letters*, vol. 27, no. 8, pp. 861–874, 2006.

[18] *Gerrit*, https://gerrit.googlesource.com/gerrit/, Accessed: 2022-05-29.

[19] *GitHub Copilot*, https://copilot.github.com/, Accessed: 2022-02-20.

[20] Glosser.ca, *Derivative of File:Artificial neural network.svg*, https://commons.wikimedia.org/w/index.php?curid=24913461, Accessed: 2022-06-13.

[21] *GloVe: Global Vectors for Word Representation*, https://github.com/stanfordnlp/GloVe, Accessed: 2022-02-20.

[22] I. Gomes, P. Morgado, T. Gomes, and R. Moreira, "An overview on the static code analysis approach in software development," *Faculdade de Engenharia da Universidade do Porto, Portugal*, 2009.

[23] R. Guo, I. Simpson, T. Magnusson, C. Kiefer, and D. Herremans, "A variational autoencoder for music generation controlled by tonal tension," *arXiv preprint arXiv:2010.06230*, 2020.

[24] A. E. Hassan, "Predicting faults using the complexity of code changes," *2009 IEEE 31st international conference on software engineering*, pp. 78–88, 2009.

[25] G. E. Hinton, "Deep belief networks," *Scholarpedia*, vol. 4, no. 5, p. 5947, 2009.

[26] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the national academy of sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.

[27] X. Huo and M. Li, "Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code.," *IJCAI*, pp. 1909–1915, 2017.

[28] Y. Kamei, E. Shihab, B. Adams, *et al.*, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[29] Q. Ke, J. Liu, M. Bennamoun, S. An, F. Sohel, and F. Boussaid, "Computer vision for human–machine interaction," *Computer Vision for Assistive Healthcare*, pp. 127–145, 2018.

[30] *Keras documentation*, https://keras.io/api/layers/, Accessed: 2022-07-15.

[31] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on software engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[32] Y. Kim, "Convolutional neural networks for sentence classification," *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1746–1751, Oct. 2014.

[33] H.-Y. Li, S.-T. Shi, F. Thung, *et al.*, "Deepreview: Automatic code review using deep multi-instance learning," *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pp. 318–330, 2019.

[34] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.

[35] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing*, vol. 27, pp. 504–518, 2015.

[36] C. D. Manning and P. Raghavan, "Schü tze h," *Introduction to information retrieval*, vol. 3, 2008.

[37] *Medium - logistic regression vs SVM*, https://medium.com/axum-labs/logistic-regression-vs-support-vector-machines-svm-c335610a3d16, Accessed: 2022-06-13.

[38] *Medium convolution*, https://medium.com/@bdhuma/6-basic-things-to-know-about-convolution-daef5e1bc411, Accessed: 2022-06-13.

[39] G. Menardi and N. Torelli, "Training and assessing classification rules with imbalanced data," *Data mining and knowledge discovery*, vol. 28, no. 1, pp. 92–122, 2014.

[40] T. Mikolov, "Language modeling for speech recognition in czech," Ph.D. dissertation, Masters thesis, Brno University of Technology, 2007.

[41] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[42] T. Mikolov, J. Kopecky, L. Burget, O. Glembek, *et al.*, "Neural network based language models for highly inflective languages," *2009 IEEE international conference on acoustics, speech and signal processing*, pp. 4725–4728, 2009.

[43] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[44] *Mythic AI principle*, https://mythic.ai/technology/analog-computing/, Accessed: 2022-06-13.

[45] T. Paine, H. Jin, J. Yang, Z. Lin, and T. Huang, "Gpu asynchronous stochastic gradient descent to speed up neural network training," *arXiv preprint arXiv:1312.6186*, 2013.

[46] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543, 2014.

[47] D. M. Powers, "Evaluation: From precision, recall and f-measure to roc, informedness, markedness and correlation," *arXiv preprint arXiv:2010.16061*, 2020.

[48] R. Purushothaman and D. E. Perry, "Toward understanding the rhetoric of small source code changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 511–526, 2005.

[49] *Python 3.6 Docs*, https://docs.python.org/3.6/, Accessed: 2022-05-29.

[50] *Python Requests library*, https://pypi.org/project/requests/, Accessed: 2022-05-29.

[51] A. Rajaraman and J. D. Ullman, "Mining of massive datasets: Data mining (ch01)," *Min. Massive Datasets*, vol. 18, pp. 114–142, 2011.

[52] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[53] J. Schmidhuber, S. Hochreiter, *et al.*, "Long short-term memory," *Neural Comput*, vol. 9, no. 8, pp. 1735–1780, 1997.

[54] S.-T. Shi, M. Li, D. Lo, F. Thung, and X. Huo, "Automatic code review by learning the revision of source code," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 4910–4917, 2019.

[55] R. Tronci, G. Giacinto, and F. Roli, "Dynamic score combination: A supervised and unsupervised score combination method," *International Workshop on Machine Learning and Data Mining in Pattern Recognition*, pp. 163–177, 2009.

[56] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[57] C. Xiao, H. Zhong, Z. Guo, *et al.*, "Cail2018: A large-scale legal dataset for judgment prediction," *arXiv preprint arXiv:1807.02478*, 2018.

[58] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," *2015 IEEE International Conference on Software Quality, Reliability and Security*, pp. 17–26, 2015.

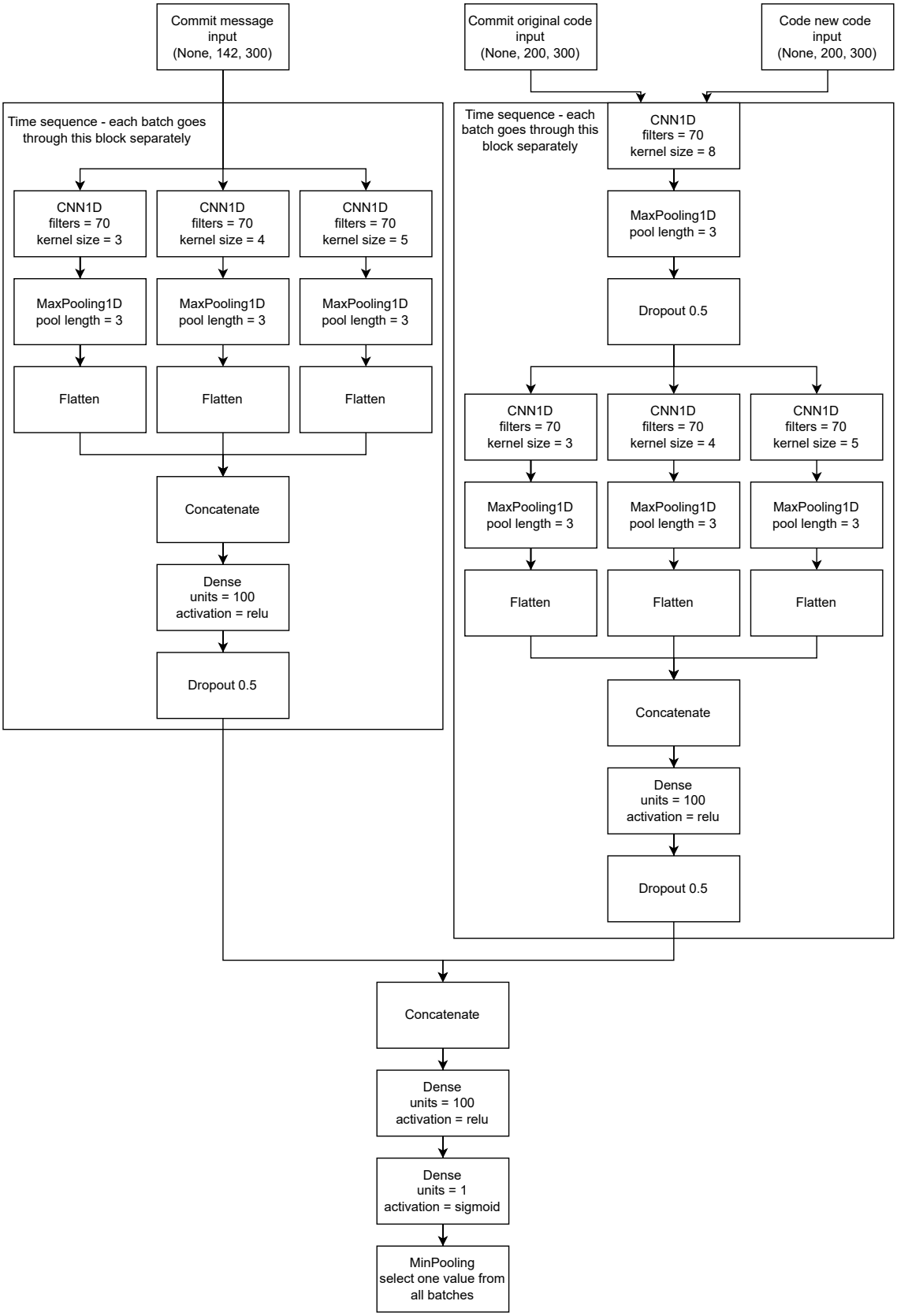
[59] W. J. Youden, "Index for rating diagnostic tests," *Cancer*, vol. 3, no. 1, pp. 32–35, 1950.

[60] H. Zhang, M. Gu, X. Jiang, *et al.*, "An optical neural chip for implementing complex-valued neural network," *Nature Communications*, vol. 12, no. 1, pp. 1–11, 2021.

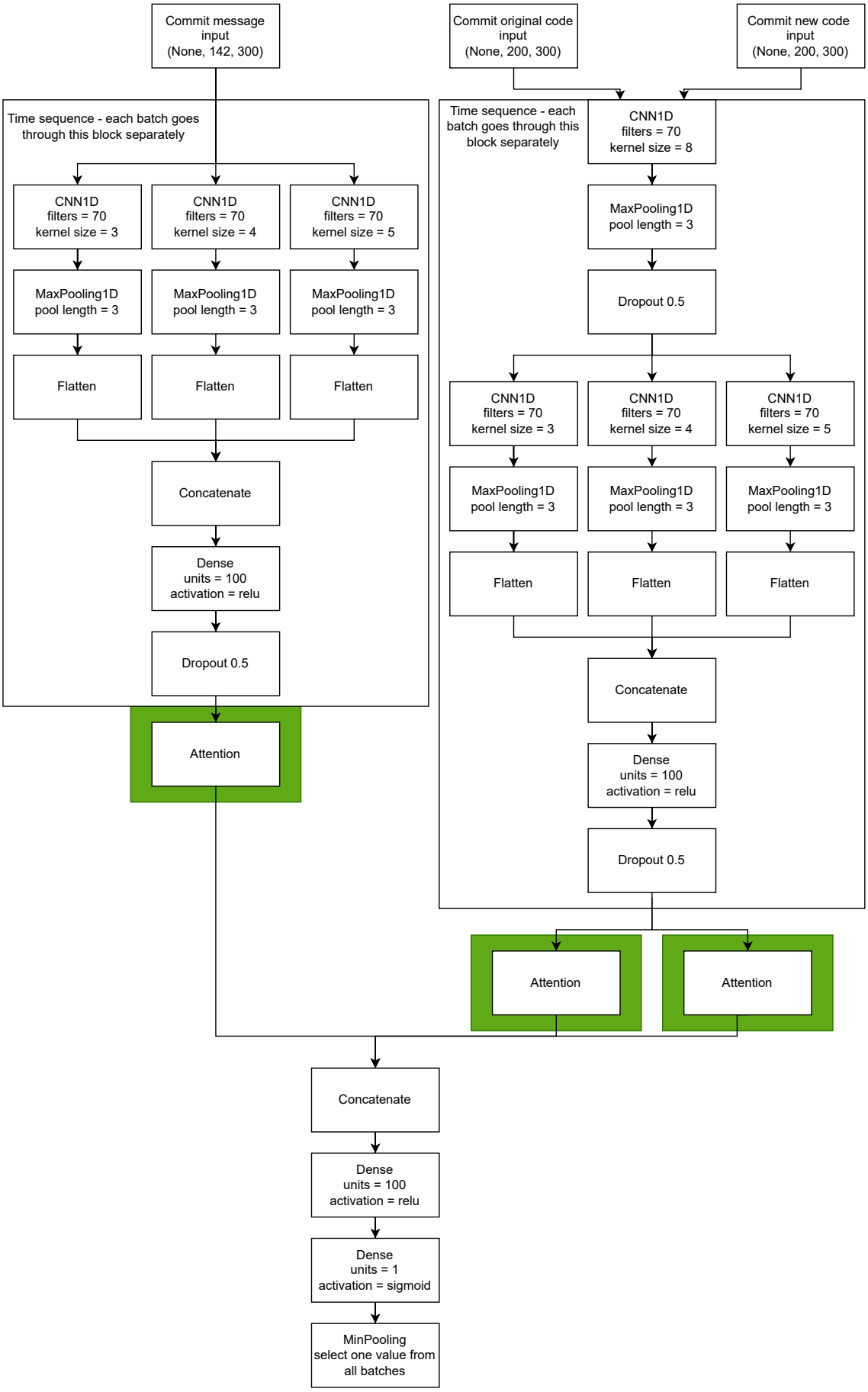
[61] Y. Zhang, W. Zheng, and M. Li, "Learning uniform semantic features for natural language and programming language globally, locally and sequentially," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 5845–5852, 2019.

A Attachments

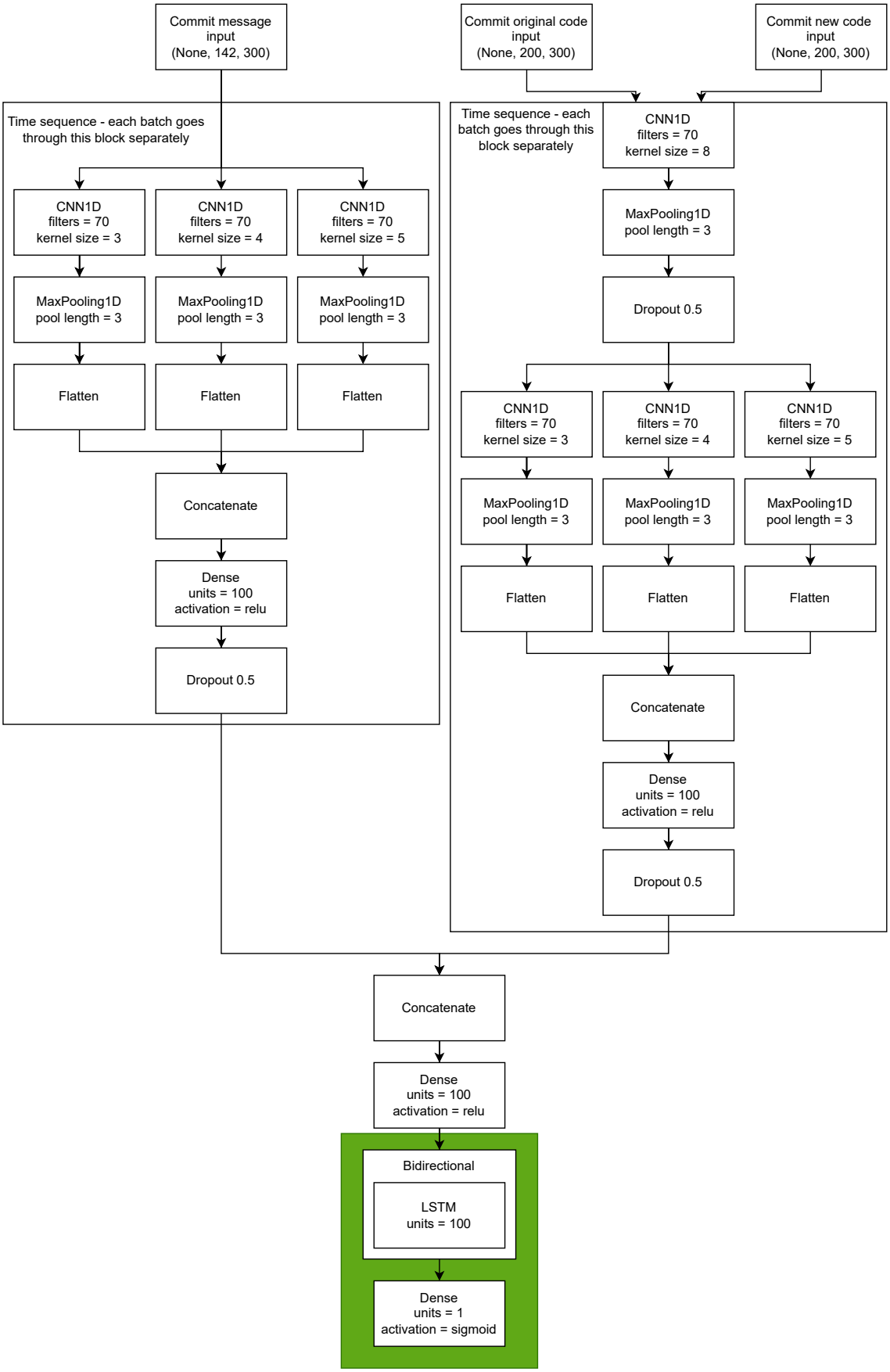
A.1 Block diagrams



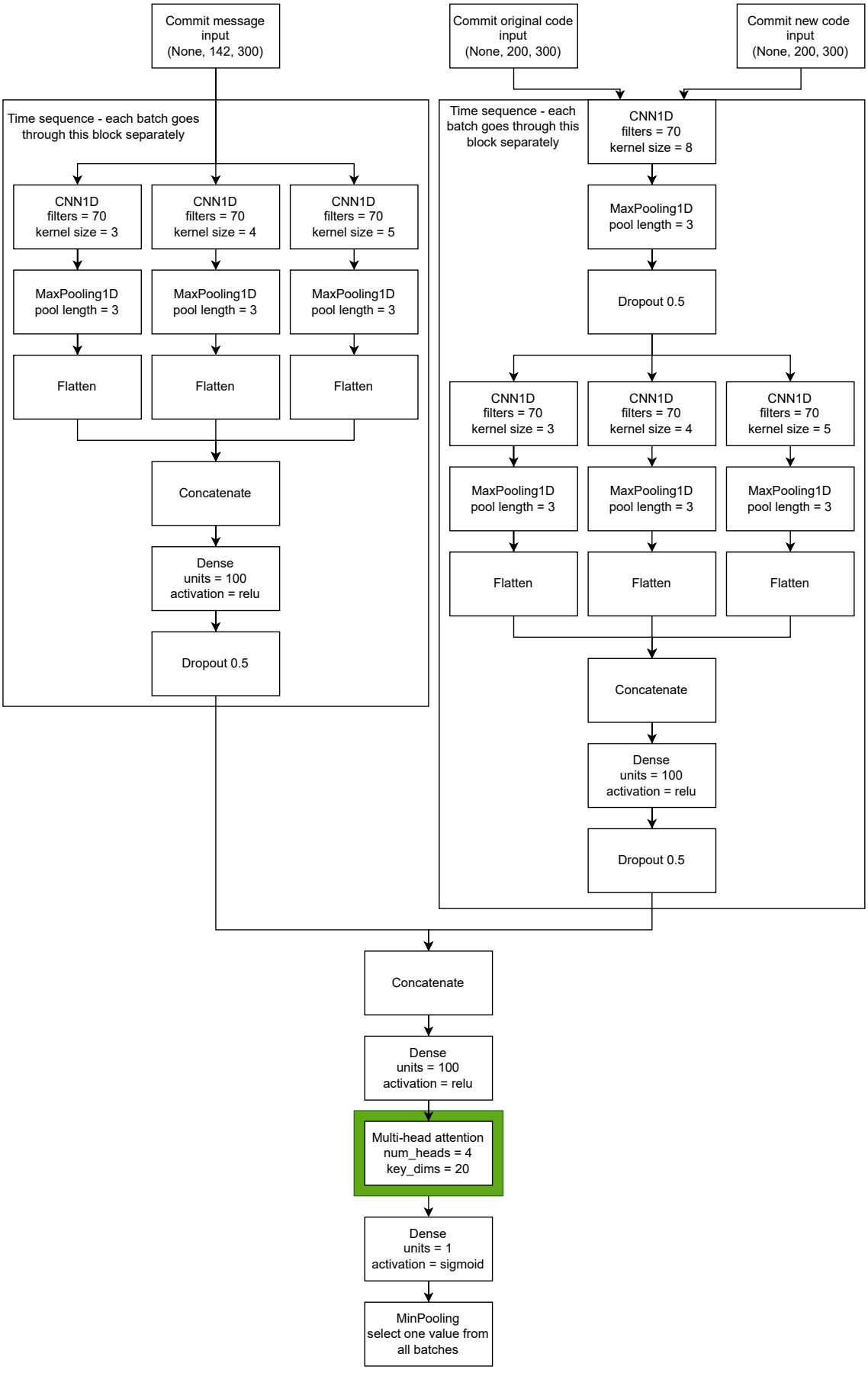
Attachment A.1: Block diagram of deepreview model constructed according to [33].



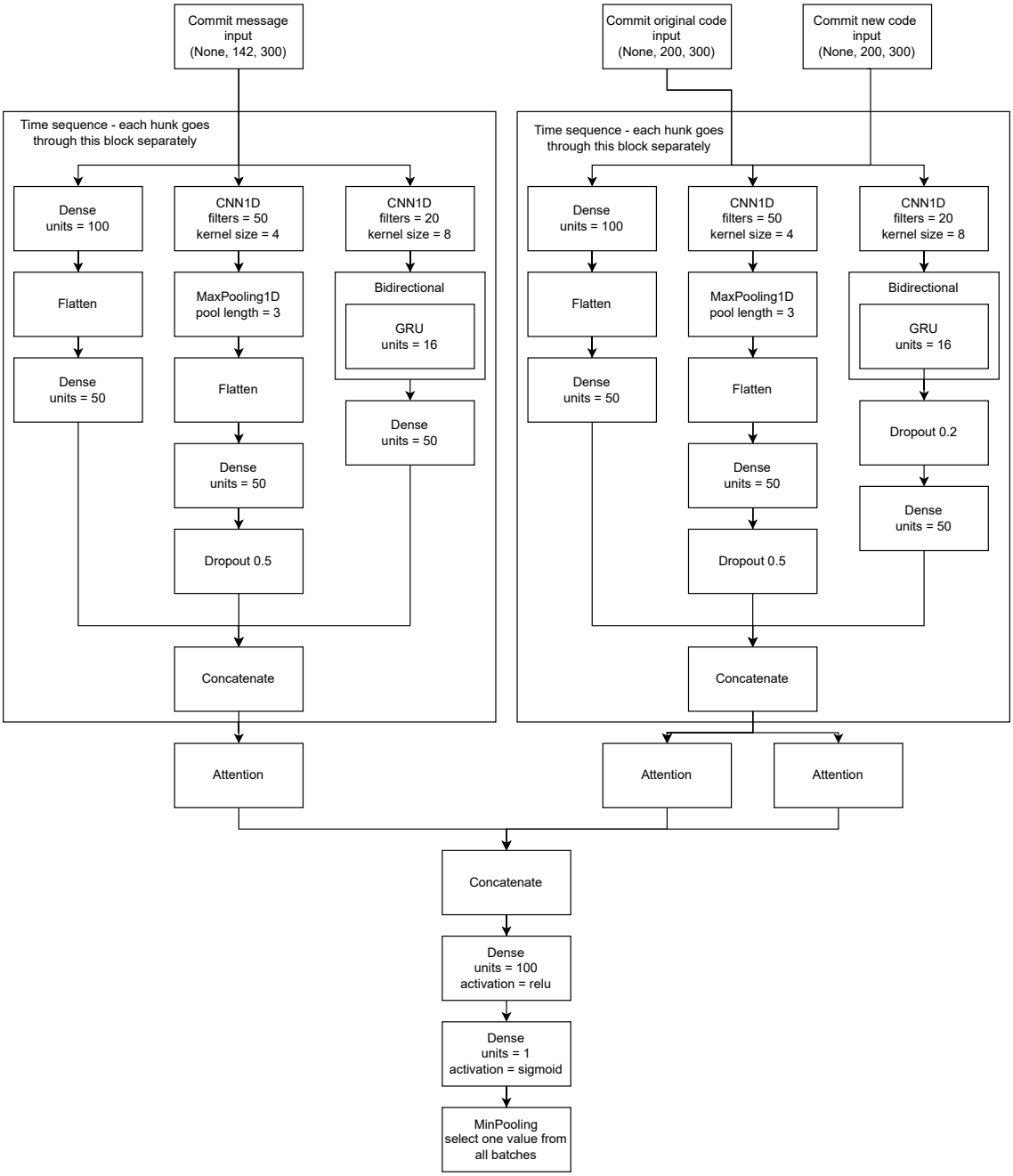
Attachment A.2: Block diagram of deepreview model with added attention layers (green boxes).



Attachment A.3: Block diagram of deepreview model with added LSTM layer (green box).

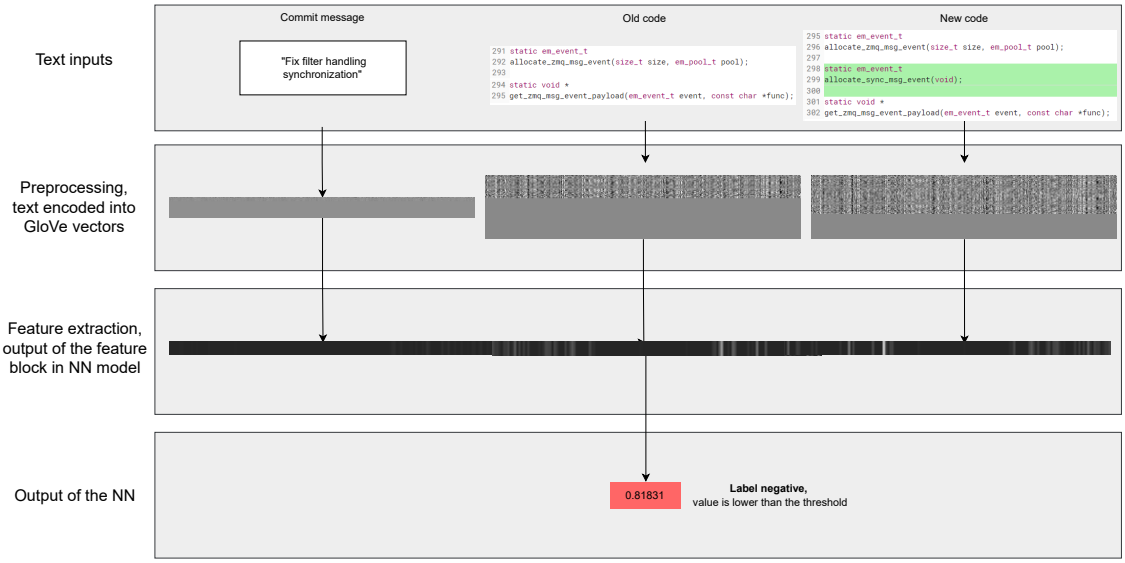


Attachment A.4: Block diagram of deepreview model with added Multi-head attention layer (green box).

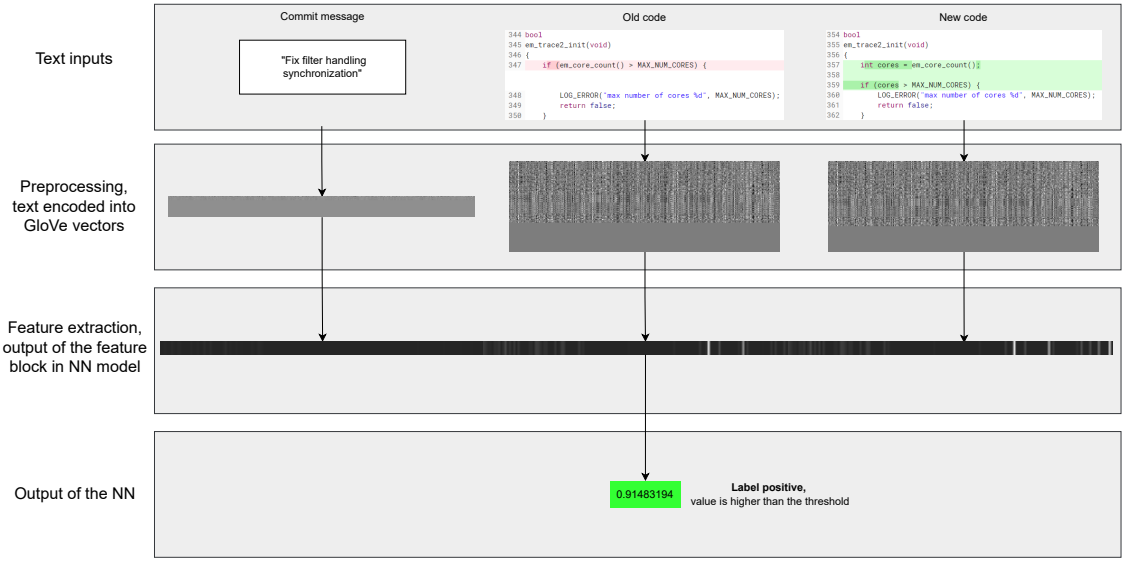


Attachment A.5: Block diagram of model with 3 types of features.

A.2 Schemes



Attachment A.6: Scheme of single hunk flowing through the 3 types of features model and achieve negative label.



Attachment A.7: Scheme of single hunk flowing through the 3 types of features model and achieve positive label.