

A Case for Deep Learning in Mining Software Repositories

H.L.D. Nijessen

Delft University of Technology



A Case for Deep Learning in Mining Software Repositories

by

H.L.D. Nijessen

in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on 10 November, 2017 at 15:00.

Student number:	4152263	
Supervisor:	Dr. G. Gousios,	TU Delft
Thesis committee:	Prof. dr. A. van Deursen,	TU Delft
	Dr. C. Hauff,	TU Delft

An electronic version of this thesis is available at <http://repository.tudelft.nl/>.

Abstract

Repository mining researchers have successfully applied machine learning in a variety of scenarios. However, the use of deep learning in repository mining tasks is still in its infancy. In this thesis, we describe the advantages and disadvantages of using deep learning in mining software repository research and demonstrate these by doing two case studies on pull requests. In the first, we train neural models to predict, on arrival, whether a pull request is going to be merged or not. In the second, we train neural models to answer the question: given two pull requests, are these similar? We show that using neural models, researchers are able to avoid feature engineering, because these models can be trained on raw data. Furthermore, neural models have the potential to outperform traditional supervised machine learning models, due to being able to learn relevant features by themselves. However, the power of neural models comes at a cost: optimizing the parameters of neural models and explaining neural models is difficult and training them is costly. We, therefore, recommend researchers to take into account well performing neural architectures in other domains, such as natural language processing, before creating novel architectures. Furthermore, it is therefore important to include a less costly baseline when using neural models in research, to show that the power and thereby the cost of neural models is justified.

Preface

Approximately a year and a half ago, I walked out of Georgios Gousios' office. I had just agreed to do a thesis that involved combining deep learning with source code analysis. "But I have no knowledge of deep learning whatsoever, isn't this going to be extremely hard?" I had asked him, as I had never even followed a course about machine learning, let alone deep learning. However, I was intrigued. "I have no practical experience with deep learning either, but we will learn it together" Georgios had answered. The year that followed after the start of my thesis proved to be quite challenging, but looking back, I am happy that I took the challenge. I learned a lot and therefore would like to thank Georgios for convincing me to accept the thesis topic and for arranging the required hardware. I would also like to thank Jan-Willem Manenschijn and Olaf Maas for getting me through the hard moments with coffee breaks. Furthermore, I would also like to thank Herman Banken for the valuable feedback he has given me on my work and I want to thank my family and friends for the support they have given me during this thesis. Finally, I want to apologize to all the people to whom I said "did you know you can use deep learning to solve that?". I must have read too many interesting papers about deep learning this year.

*H.L.D. Nijessen
Delft, November 2017*



Machine Learning, xkcd ¹

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Research question & approach	2
1.3	Outline	3
2	Background and related work	5
2.1	Pull requests	5
2.1.1	Anatomy of a pull request	5
2.2	Deep learning	7
2.2.1	Feedforward neural network	8
2.2.2	Recurrent neural network	8
2.2.3	Convolutional neural network	9
2.2.4	Siamese neural network	10
2.2.5	Training neural networks	10
2.3	Deep learning in natural language processing	11
2.3.1	Word2Vec	11
2.3.2	Doc2Vec	11
2.4	Deep learning in repository mining research	12
3	Predicting pull request merge decisions	13
3.1	Data	13
3.1.1	Pre-processing	14
3.2	Models	14
3.2.1	Model 1	14
3.2.2	Model 2	15
3.2.3	Random forest baseline	15
3.3	Training	16
3.3.1	Model 1 & Model 2	16
3.3.2	Random forest baseline	16
3.4	Experiments	16
3.5	Results	17
3.6	Conclusion	17
4	Finding similar pull requests	19
4.1	Data	19
4.1.1	Construction	19
4.1.2	Evaluation	21
4.1.3	Analysis	22
4.2	Preprocessing	23
4.2.1	Tokenizing	23
4.2.2	Doc2vec & Word2vec	23
4.2.3	Bag-of-words	24
4.3	Models	24
4.3.1	Feedforward neural network	24
4.3.2	Siamese convolutional neural network	25
4.3.3	XGBoost	26
4.3.4	SVM	26

4.4	Experiments	26
4.4.1	Experiment 1	26
4.4.2	Experiment 2	26
4.4.3	Experiment 3	27
4.4.4	Experiment 4	27
4.4.5	Experiment 5	28
4.5	Results	28
4.5.1	Experiment 1	28
4.5.2	Experiment 2	28
4.5.3	Experiment 3	29
4.5.4	Experiment 4	29
4.5.5	Experiment 5	29
4.6	What did the models learn	30
4.7	Threats to validity	32
4.8	Conclusion	33
5	Lessons learned	35
5.1	The cost of training neural models	35
5.2	Optimizing neural models	35
5.3	Explaining neural models	36
6	Future work	39
6.1	Predicting pull request merge decisions	39
6.2	Finding similar pull requests	39
6.3	New research ideas	39
7	Conclusion	41
	Bibliography	43

1

Introduction

Traditionally, research on neural networks was confined by hardware limitations and the absence of large labeled datasets. In the recent years, advances in hardware, mainly Graphics Processor Units (GPUs), along with the availability of large labeled datasets such as ImageNet [1], have enabled researchers to train neural networks at an unprecedented scale, mainly by stacking more learning layers in sequence. Multiple variations of these so-called Deep Neural Networks (DNNs) [2] have been successfully applied on high dimensional data ranging from images to music and, by uncovering latent non-linear relationships, they have achieved record breaking performance in tasks such as image classification [3] and speech recognition [4]. All of these variations fall under the umbrella term *deep learning*. Recently, deep learning is also being applied with increasing success in research fields such as Natural Language Processing (NLP), special cases of which are also of interest to software engineering researchers [5, 6].

A particularly interesting feature of DNNs is that they can consume raw data (or data with minor pre-processing), such as text, images or bit sequences and infer the discriminating characteristics automatically, as part of their training step. In traditional supervised machine learning, the vast majority of the researchers' time and creative effort is spent on cleaning up the data and inventing or inferring features that can be used to predict the outcome, a task referred to as *feature engineering*. Given enough diverse data, DNNs help researchers in uncovering interesting relationships in the data without resorting to feature engineering; they may even help in uncovering relationships that humans may not foresee.

Figure 1.1¹ shows a comic published by XKCD on the 24th of September 2014, which displays the general consensus at that time: recognizing objects in images is an extremely difficult computer science task. However, in May 2015 Google released the Google Photos² service which uses deep learning to solve this previously difficult task at scale. Nowadays even a television company is able to create an image classifier with high accuracy with ease³. These examples show how the power of deep learning is changing the world.

1.1. Motivation

The Mining Software Repositories (MSR) research field has as goal extracting valuable information from the rich data that can be found in software repositories, for example GitHub⁴ and other relevant datasources such as bugtrackers, documentation and mailing lists, but also fora discussing software engineering such as Stack Overflow⁵. Using this information, researchers can empirically investigate the software development process and thereby validate software development methodologies, discover trends and develop tools that improve various facets of software development. The main goal of this research field is to make software development easier and to improve the quality of developed software.

¹<https://xkcd.com/1425/>

²<https://photos.google.com/>

³<https://medium.com/@timanglade/ef03260747f3>

⁴<https://www.github.com/>

⁵<https://www.stackoverflow.com/>



Figure 1.1: Comic published by XKCD on the 24th of September 2014

The datasources used in MSR research mostly consist of unstructured data, such as source code and text. This data has to be processed before it can be analyzed. To be able to extract information from this unstructured data, repository mining researchers have successfully applied machine learning in a variety of scenarios. However, the use of DNNs in repository mining tasks is still in its infancy.

DNNs have shown promising results in other research fields. DNNs can be trained on raw data instead of requiring researchers to come up with features. Not only does this save researchers from lots of effort to find the right features. DNNs may discover features the researcher did not foresee or features we as humans cannot describe or understand. This may result in models that significantly outperform the current state of the art, as has been shown in the computer vision field in 2012 [3]. Not only can this result in better performing models, but the trained DNN can also serve as a measure for the amount of relevant information a piece of data contains.

Additionally, DNNs can make applications possible that have not been possible before. A great example of this is the ease and accuracy of which it is nowadays possible to recognize objects in images [7] and even generate new images based on a description [8]. A good example in repository mining research is the paper by Allamanis et al. [9] where they use DNNs to summarize the functionality of a method based on source code. Furthermore, DNNs open up the possibility for research on unstructured artifacts, such as diagrams. In this thesis, we attempt to make repository mining researchers aware of the promises and perils of using DNNs in repository mining research.

Due to a large part of the data relevant to repository mining researchers consisting of natural language and Hidle et al. [10] showing that source code, like natural languages, is likely to be repetitive and predictable, we take inspiration from the NLP research field for the neural models we use in this thesis.

1.2. Research question & approach

Given the context described in section 1.1, this thesis serves to answer the research question:

RQ What are the benefits and perils of using deep learning in repository mining research?

We answer the research question stated above by performing two case studies on pull requests submitted on GitHub. The first study is a replication of the existing research by Gousios et al. [11] in which the authors try to predict whether a pull request, on arrival, is going to be merged or not. In this replication we show that our DNN is able to extract more information from raw data than the original model using engineered features based on the same data. The second is a new research in which we

use DNNs to automatically collect labeled training data and train DNNs to answer the question: *given two pull requests, are these similar or not*. The two case studies serve as examples of how DNNs can be incorporated into repository mining research.

1.3. Outline

This thesis consists of 7 chapters. We first describe the required background information and related work in chapter 2. In chapter 3 we then compare deep learning with manual feature engineering in solving the merge prediction task. Subsequently, in chapter 4 we use deep learning techniques to create classifiers that are able to recognize whether two PRs are similar or not. In chapter 5 we explain, based on our experiences during this thesis, some disadvantages one has to keep in mind when considering the use of neural models. In chapter 6 we discuss future work and finally, in chapter 7 we finish this thesis with a conclusion.

2

Background and related work

In this section we explain the required background knowledge and describe the related work. We first explain the concept of pull requests. Then we explain the deep learning technology we use in this thesis and describe how deep learning is being used in NLP and the MSR research field.

2.1. Pull requests

Pull-based development is becoming increasingly popular as a distributed software paradigm. Especially open-source projects are embracing it by migrating to (social) repository hosting services such as GitHub and Bitbucket [12]. These services allow any user to fork a public repository, which creates a new public repository owned by this user. The user can then modify the repository and submit the changes to the original repository. This submission is called a pull-request (PR). Repository owners can then either decide to accept the PR by merging it, discuss the PR by placing a comment or reject the PR by closing it.

When submitting a PR on GitHub, in addition to the commits that contain the changes the PR is making, users are asked to provide a title and optionally a description. Other users can discuss the PR by leaving comments.

Gousios et al. [13] show that 53% of non-merged PRs are rejected for reasons related to the distributed nature of pull-based development, such as not following the PR conventions of a project or implementing a feature to project not (yet) needs. Only 13% of the PRs are not merged because of technical reasons. Furthermore, they conclude that the merge or not merge decision is mainly affected by whether the pull request modifies recently modified code.

Kalliamvakou et al. [14] mention that only a fraction of projects uses pull requests. Furthermore, most pull requests appear as non-merged even if they are merged, since not all PRs are merged using the GitHub interface. PRs that are not merged using the GitHub interface, are not recognized as merged by GitHub. Therefore they recommend the use of a list of heuristics to check if a PR is merged or not.

van der Veen et al. [15] mention that repository owners have to invest lots of time to manage open PRs and that GitHub is lacking tools to effectively prioritize them. They therefore use traditional machine learning techniques to automatically prioritize PRs.

2.1.1. Anatomy of a pull request


When submitting a PR on GitHub users are asked to provide a title that describes the modification the user is submitting in a few words. Optionally, a description can be added that describes the PR in more detail. This description is written in Markdown and can thus contain simple markup and images. Furthermore, it can contain references to issues and other PRs as shown in figure 2.1.

Every GitHub user is able to comment on a PR. This can for example be used to suggest improvements, request more explanation or explain why a PR is not going to be merged. Like the description, comments are written in Markdown. An example of a GitHub discussion is shown in figure 2.2.


The modifications of a PR can be submitted in one or multiple git commits. Based on these commits a *unified diff patch file*¹ is generated, which shows the changes the PR is making. An example of such

¹<https://git-scm.com/docs/diff-format>

Add short description about Dense and Activation #8316

 **Open** sauerburger wants to merge 1 commit into fchollet:master from sauerburger:info_dense_activation_2

Conversation 0 Commits 1 Files changed 1

 sauerburger commented 2 days ago First-time contributor

Add a paragraph to the 'Getting started with the Keras Sequential model' section explaining how a Dense layer looks like and how it relates to activation functions.

See issue [#8179](#) and related PR [#8178](#).





  Add short description about Dense and Activation Verified ✓ be64eae

Figure 2.1: Example of a submitted PR.



 TimZaman commented 4 days ago Contributor

Lol, duplicate of [#8285](#) ! Merge whichever, they are practically feature compatible.

 bzamecnik commented 4 days ago Contributor

@TimZaman Aah :) In the morning I've just made a PR from the branch that was waiting there for a week since at night I managed to make a working use case for that. Seems you made a PR earlier in the morning :)

The PRs are basically about the same thing. When looking at your commit, passing `feed_dict` via `session_kwargs` is nicely uniform, but it's not usable in `model.compile()`, since we'd provide constant values ([8593309](#)).

  TimZaman commented 4 days ago Contributor

I'm in Mountain View; did the change last night in this TZ. The feed dict dictionary passed in is not (deep)copied, so you can still change the feed dict contents? That would make a use case. It's up to the user anyway.

Figure 2.2: Example of a discussion about a PR that contains references to another PR.

a diff file is shown in Listing 1.

```
diff --git a/keras/callbacks.py b/keras/callbacks.py
index 42f1573c83..d42ea589e7 100644
--- a/keras/callbacks.py
+++ b/keras/callbacks.py
@@ -399,7 +399,7 @@ def on_epoch_end(self, epoch, logs=None):
     self.epochs_since_last_save = 0
     filepath = self.filepath.format(epoch=epoch, **logs)
     if self.save_best_only:
-         current = logs.get(self.monitor)
+         current = logs.get(self.monitor)[-1]
         if current is None:
             warnings.warn('Can save best model only with %s available, '
                             'skipping.' % (self.monitor), RuntimeWarning)
@@ -488,7 +488,7 @@ def on_train_begin(self, logs=None):
     self.best = np.Inf if self.monitor_op == np.less else -np.Inf

     def on_epoch_end(self, epoch, logs=None):
-         current = logs.get(self.monitor)
+         current = logs.get(self.monitor)[-1]
         if current is None:
             warnings.warn('Early stopping requires %s available!' %
                             (self.monitor), RuntimeWarning)

diff --git a/keras/utils/generic_utils.py b/keras/utils/generic_utils.py
index 76477d5ac3..5d224d7492 100644
--- a/keras/utils/generic_utils.py
+++ b/keras/utils/generic_utils.py
@@ -289,10 +289,16 @@ def update(self, current, values=None, force=False):
     info += ' - %s:' % k
     if isinstance(self.sum_values[k], list):
         avg = self.sum_values[k][0] / max(1, self.sum_values[k][1])
-         if abs(avg) > 1e-3:
-             info += ' %.4f' % avg
+         if isinstance(avg, float):
+             if abs(avg) > 1e-3:
+                 info += ' %.4f' % avg
+             else:
+                 info += ' %.4e' % avg
         else:
-             info += ' %.4e' % avg
+             if abs(avg[-1]) > 1e-3:
+                 info += ' %.4f' % avg[-1]
+             else:
+                 info += ' %.4e' % avg[-1]
     else:
         info += ' %s' % self.sum_values[k]
```

Listing 1: Example of a diff file.

A diff file contains a segment for each file that is changed. This segment starts with information about the file being changed, such as the original file name and, when a file is renamed, the new file name.

Subsequently, the changes to the contents of the file are reported in the form of *hunks*, where a hunk shows an area where the file is changed. A hunk starts with a hunk header, which contains information about the location of the changes in the file. This header is indicated by a line that starts with "@@". After this hunk header, the changes are reported. A line that starts with a single "-" denotes a removed line, while a single "+" denotes an added line. When a line is partially changed, this is shown by the removal of the original line and addition of a new line including the change. The neighboring lines of an added or removed line are shown to provide context. These lines do not start with a symbol.

2.2. Deep learning

In supervised machine learning, instead of programming a computer manually, it is shown examples of inputs and outputs using which it learns how to produce the outputs based on those inputs by creating a statistical model. Traditionally, the process of training a machine learning model consists of a feature engineering step. During this step, the researcher or engineer manually devises and extracts features from the raw data that they think are predictive for the problem the machine learning model is planned to solve. These features are then fed as input to the machine learning model, which learns to recognize

patterns in these features. Feature engineering is considered difficult, time-consuming and requires domain knowledge.

Deep learning takes a different approach: a neural network consisting of multiple layers is trained on raw data. The first layer, given as input the raw data, learns to extract low-level features, such as the edges in an image. The next layer combines these low level features forming higher-level features. Eventually, this results in later layers recognizing high-level features such as objects in images. Instead of relying on the domain knowledge of humans, deep learning models learn to recognize important features themselves skipping the feature engineering step. Besides resulting in less manual work, these models are capable of learning features that are superior to the features devised by humans resulting in better performing models compared to traditional machine learning models. The use of more layers generally results in better performing models [7], hence the name *deep learning*. Compared to traditional machine learning models, deep learning models have an enormous number of parameters. Deep learning models, therefore, in general, need larger datasets to be trained successfully.

2.2.1. Feedforward neural network

A neural network consists of neurons organized in interconnected layers that form a graph. A neuron receives one or multiple inputs and produces an output which is calculated using the following formula:

$$y = f\left(\sum_{i=1}^n a_i \cdot x_i + b\right) \quad (2.1)$$

The neuron first calculates a weighted sum of the inputs using the weights a_i for each input x_i . Then it adds a bias b and applies an activation function $f(x)$. The weights a_i and the bias b are adjustable and are learned by training the network. The activation function $f(x)$ is a non-linear differential function. The non-linearity of the activation functions ensures that the neural network is able to approximate non-linear functions. If $f(x)$ was linear, the neural network could only approximate linear functions. The differential property of $f(x)$ is used during training of the neural network. Frequently used activation functions are:

ReLU $f(x) = \max(0, x)$

tanh $f(x) = (e^x - e^{-x}) / (e^x + e^{-x})$

sigmoid $f(x) = 1 / (1 + e^{-x})$

ReLU is currently the recommended activation function [16], due to its computational efficiency and the fact that it does not suffer from vanishing gradients.

In the most simple neural model, a *feedforward neural network* (FFNN), the neurons in layer n are connected to all neurons in layer $n + 1$. Information flows from the first layer, receiving the input, through all layers to the last layer that produces an output. Instead of calculating the output of each neuron individually the output of layer n in a feedforward neural network can be calculated using the following formula:

$$y_n = f(W_n \cdot y_{n-1} + b_n) \quad (2.2)$$

Here $y_n \in \mathbb{R}^k$ and $y_{n-1} \in \mathbb{R}^j$ are vectors representing the output of respectively layer n and $n - 1$ containing k and j neurons, $W_n \in \mathbb{R}^{k \times j}$ is a matrix containing all weights of the neurons in layer n , $b_n \in \mathbb{R}^k$ is a vector containing all biases of the neurons in layer n and $f(x)$ is the activation function. A feedforward neural network learns to approximate a function $y = f(x)$ where x and y are fixed size vectors. The tunable hyperparameters of this model are the number of layers, the number of neurons for each layer and the activation function.

2.2.2. Recurrent neural network

Feedforward neural networks are only able to handle inputs of fixed size and produce outputs of fixed size. Instead of accepting one input of fixed size, Recurrent neural networks (RNNs) step over a sequence of inputs of fixed size and produce an output during each step, resulting in a sequence of outputs. Each layer in an RNN cycles its output back into that same layer which uses it in the next

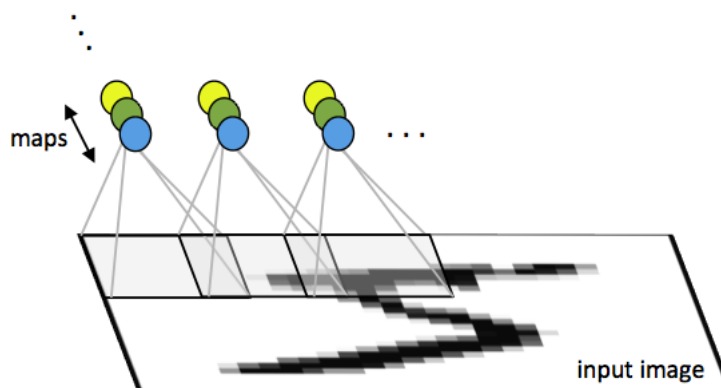


Figure 2.3: Convolutional layer sliding over an image. Units of the same color have tied weights and units of different color represent different filter maps.²

step. This acts as a memory of not only the previous step, but all steps before that. The output of a layer in an RNN can thus be calculated using the following adaption of equation 2.1:

$$y_{t,n} = f(W_n \cdot y_{t,n-1} + U_n \cdot y_{t-1,n} + b_n) \quad (2.3)$$

Here $y_{t,n} \in \mathbb{R}^k$ is the output of layer n during step t . $U_n \in \mathbb{R}^{k \times k}$ contains the weights of the feedback loop of layer n . RNNs can be used in multiple modes:

One to many By repeatedly feeding an RNN the same input and collecting each output.

Many to one By feeding an RNN a sequence of inputs and collecting only the last output.

Many to many By feeding an RNN a sequence of inputs and collecting each output.

Vanilla RNNs are not used in practice; the Long short-term memory (LSTM) and Gated Recurrent Unit (GRU) [17], are improved implementations of an RNN using a slightly different architecture. Both have been shown to perform comparable, but the GRU is slightly more efficient due to using fewer calculations.

The tunable hyperparameters of the GRU and LSTM are the number of layers and the number of neurons for each layer. The GRU and LSTM both have fixed activation functions as part of their architecture.

2.2.3. Convolutional neural network

Convolutional Neural Networks [3, 18] (CNNs) have initially been developed as a more efficient way of processing images compared to using regular feedforward neural networks, since these do not scale well when used for image processing. A convolutional layer consists of multiple so-called *filters* that process the pixels of an image by sliding over parts of it. For each part of the image, an output is generated. These filters, sometimes also called *feature maps*, learn to recognize location invariant features. By stacking multiple convolutional layers, models can be created that learn to recognize high-level features in images [3, 7, 19]. Figure 2.3 illustrates a convolutional layer with as input an image.

After the success in image processing, CNNs have been shown to be successful in multiple NLP tasks [20, 21]. In the context of NLP, CNNs can be interpreted as learning to recognize the most relevant n-grams in sequences in a more efficient manner than n-gram models do. Instead of sliding over the pixels of an image, in NLP a filter slides over words or characters of a sentence. The output of a convolutional layer can be calculated using the following equation:

$$y_n = f(W_n \cdot y_{n-1}[i - flen : i] + b_n) \quad (2.4)$$

²<http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/>

Here y_n is the output of layer n and consists of a sequence of vectors. In the first layer, each vector $v \in \mathbb{R}^{emb}$ is a vector representation of a word or a character. W is a tensor of rank-3 $W \in \mathbb{R}^{fnum,flen,emb}$ which contains trainable weights, where $fnum$ and $flen$ are the number of filters and the length of these filters respectively. f is a non-linear activation and $b \in \mathbb{R}^{fnum}$ is a trainable bias. Kim [22] proposes a simple general purpose CNN architecture that can be used for NLP tasks, which we use in this thesis. An advantage of this model compared to an LSTM or GRU is that CNNs process a sequence in parallel, whereas LSTMs and GRUs do this sequentially. Empirical research [23] has shown that both have comparable performance: LSTMs and GRUs perform better when the whole sequence or a long-range semantic dependency is important for making a prediction. On the other hand, CNNs perform well on tasks that require identification of important key parts of a sequence. The tunable hyperparameters of a convolutional layer in NLP are: the number of filters per layer and the length of each filter. Multiple convolutional layers can be stacked on top of each other, but Kim only uses one layer.

2.2.4. Siamese neural network

A siamese neural network [24] is a neural network that has two identical subnetworks which accept distinct inputs and are joined later in the network. These subnetworks share the same architecture and weights. Such an architecture can, for example, be used to compare two inputs. This architecture has two favorable properties [25]:

- Two similar inputs are not mapped to completely different locations in vector space, since they are processed by the same function.
- The network is symmetric, such that two inputs result in the same output independent of in which order the inputs are presented.

2.2.5. Training neural networks

During training, a neural network is shown a training set containing inputs labeled with the output the neural network should produce. The neural network takes each input and calculates an output by feeding it through all layers. Then a loss function calculates the error between the output the neural network should produce and actual output. For each weight and bias in the neural network, the partial derivative is calculated with respect to the loss function. Using these partial derivatives, the biases and weights are adjusted to minimize the loss function. This algorithm is called *stochastic gradient descent* (SGD). However, calculating all these partial derivatives individually is inefficient. Therefore an optimized implementation, called *backpropagation* [26], is used that uses the chain rule to propagate the partial derivatives from the end of the neural network to the beginning. During training a neural network is shown the training set multiple times. Such a training round is called an epoch.

Multiple improvements have been made based on the standard SGD, resulting in numerous co-existing advanced optimizing algorithms. These improvements have shown to make neural networks learn faster, reach better results and be more robust with respect to the choice of hyperparameters [27]. Which algorithm to choose depends on the dataset, the neural architecture and the task the neural network has to solve. Therefore, most of the time this choice is based on experience and intuition, as is the case with choosing many hyperparameters while training neural networks. In this work we use the RMSprop [28] and Adam [29] algorithms based on many research papers that came before us [30, 31].

Cross validation is a commonly used validation technique to assess the quality of a statistical model. The main reason for using cross validation is to ensure that enough data is available to train and test the model. Due to the large datasets on which neural networks are trained and the fact that neural networks are very costly to train, cross validation is considered unnecessary. Instead, a training-validation-test split is used. Hyperparameter optimization is done by training on the training set and assessing the performance of the network on the validation set. After choosing the right hyperparameters, the network is tested on the test set. Apart from choosing the hyperparameters, the validation set can also be used to decide when to stop training, which is called *early stopping*. In that case, training is stopped when performance has not improved, measured on the validation set, for X epochs. In both cases, the validation set is used to prevent using knowledge about the test set to optimize the network.

Due to the large number of trainable parameters, neural network sometimes memorize the data in a training set instead of learning to generalize. This problem is called overfitting and can be caused

by using too many neurons, using too many layers or having not enough training data. Dropout [32] partially solves this problem by turning each neuron in a layer off with probability p during training. When a neuron is turned off it is temporarily removed from the network. This prevents neurons from co-adapting, because neurons cannot rely on other neurons in the same layer being active; They therefore learn more robust feature representations. Using a higher probability p results in a longer training time but can lead to better results. One therefore has to find a compromise between training time and results. Furthermore, the capacity of a neural network is reduced by using dropout. The authors of [32] advise setting the number of neurons in a layer to n/p where n is the estimated optimal number of neurons in that layer and to increase the learning rate of the optimizing algorithm by 10 to 100 times compared to the optimal learning rate for a standard neural network. Typical values of p range from 0.2 to 0.5.

2.3. Deep learning in natural language processing

An immensely large amount of research has been done on using deep learning in natural language processing (NLP). Neural models have been trained to solve problems such as sentiment analysis [33, 34], machine translation [35, 36] and question answering [37, 38]. Often, these models are general purpose models that in essence accept as input a list of vectors and generate as output a decision, a feature vector representation or another sequence of items. In this thesis we use two preprocessing techniques, Word2Vec and Doc2Vec, that have been successfully used to solve NLP problems. In this section we describe these techniques.

2.3.1. Word2Vec

Word2Vec [39] is a commonly used unsupervised pre-training method in NLP which maps words to vectors, also called *word embeddings*. Each word is placed in the vector space such that words that have the same or similar meaning are placed close together. The learned vectors have been shown to capture semantical relationships, such as the following: $Vec("King") - Vec("Man") + Vec("Woman") \approx Vec("Queen")$.

Word2Vec consists of two models: Continuous Bag Of Words (CBOW) and Skip-gram. CBOW trains a shallow neural network to predict a word given the context of that word. Skip-gram does the opposite and trains a shallow neural network to predict a context given a word. Mikolov et al [39] mention that CBOW trains faster, but Skip-gram works better for small datasets and improves accuracy on less frequent words. Both models build upon the distributional hypothesis, which states that words appearing in similar contexts, tend to have similar meaning [40].

It has been shown that pre-training with Word2Vec improves performance on multiple NLP problems [22] compared to initializing the word to vector mapping randomly and modifying it during training.

Nguyen et al. [41, 42] demonstrate that Word2Vec can also be used for mapping API elements from one programming language to another. Furthermore, Word2vec is even applicable in the context of biological sequences [43]. These results show that Word2Vec has the potential to be useful in domains other than NLP, such as code.

2.3.2. Doc2Vec

Paragraph vectors [44], better known as Doc2Vec, is an extension of Word2Vec which maps a whole document to vector space instead of individual words. The method consists of two models: Distributed Bag of Words (PV-DBOW) and Distributed Memory (PV-DM).

The PV-DBOW model trains a shallow neural network to predict randomly sampled words from a document based on the vector representing that document. This is similar to the Skip-gram model of Word2Vec. The PV-DM model is similar to the CBOW model of Word2Vec, but also uses a paragraph token, which represents the rest of the paragraph, in addition to the context of a word to predict that word. These contexts are randomly sampled from a fixed length sliding window over said paragraph.

The authors focus on modeling text in their work, but they expect Doc2Vec to be applicable to sequential data in general. It is this generalizability that motivates our choice for using Doc2Vec in this thesis.

2.4. Deep learning in repository mining research

Although it is a relatively new concept, several studies have been using deep learning in repository mining research. White et al. [45] make a case for deep learning in repository mining research by applying deep learning to the code completion task and show that their approach, using RNNs, outperforms traditional n-gram models.

White et al. [46] use a combination of RNNs and recursive neural networks to do clone detection and show that the resulting model is able to detect clones that are undetected or suboptimally reported by the tool Deckard.

Gu et al. [47] use an RNN Encoder-Decoder model, similar to those used in machine translation, to transform a natural language query into a sequence of API calls that performs the requested task. They train on a dataset that contains 7 million <API sequence, annotation> pairs collected from Java projects by pairing each first line of a javadoc comment with the code in the corresponding method.

Wang et al. [48] employ a Deep Belief Network for the task of defect prediction. They report the performance of their models on within-project defect prediction and cross-project defect prediction on a already existing dataset. They show that their models improve on both compared to the state of the art.

Gupta et al. [49] apply an RNN Encoder-Decoder model based on GRUs to fix common mistakes in the C programs written by students. Their model is able to fix 27% of the programs completely and 19% of the programs partially.

Lam et al. [50] use DNNs in combination with an information retrieval technique called rVSM to localize buggy files for bug reports. Huo et al. [51] accomplish better results on the same problem by extracting features from bug reports and code using CNNs and fusing this information using a feedforward neural layer.

Allamanis et al. [52] train neural models to suggest class, method and variable names. Similarly, Allamanis et al. [9] use CNNs for summarizing source code snippets into short descriptive function names. They use the name of a method as a summary for the code in that method.

Jiang et al. [53] train a RNN Encoder-Decoder model to generate a commit message based on a diff. They find that their model tends to generate commit messages of either very high quality or very low quality.

3

Predicting pull request merge decisions

To evaluate how the deep learning approach of avoiding the feature engineering step performs in a typical software engineering task, we conduct a benchmark between a neural network and Random Forest, a state-of-the-art non-deep baseline that offers high explainability. The task we choose is *pull request merge prediction*: at the time of a pull request arrives to a project, can we predict whether it will be merged? We choose this particular task as it has been explored several times in the past [11, 54, 55], while a human-engineered dataset is readily available [56].

We explore two models, both using a LSTM architecture: Model 1 only receives the `diff` as input (without any further processing), while Model 2 receives the `diff`, the title and the description of the pull request.

For our baseline, we employ a subset of the features described by Gousios et al. in reference [11]. To make the comparison fair, we only employ all the features that can be generated from the data that is also the input to our DNN. More elaborate features that take into account the developer's track record, the hotness of the project area that the PR is targeting or the state of the project's code base are removed, as they require data not available to the DNN.

3.1. Data

We use an updated version of the pull request dataset by Gousios and Zaidman [56] as a starting point for our experiments. The dataset contains the data of 915,000 GitHub pull requests originating from 5,543 projects in 5 languages. The dataset was compiled in late 2015.

Since more than 85% of the pull requests are merged, the dataset is inherently unbalanced; in terms of machine learning, this has the undesirable effect that most algorithms will optimize for learning the majority case (i.e., `merged`). For this reason, we choose to balance the dataset using all cases of the minority class and an equal number of randomly selected cases of the majority class.¹

To train the models, we split the data 80%/20% into a training set and a test set. For our DNNs, we split the training data once more into a training and validation set (10% of the training data) in order to minimize overfitting.

For our baseline, we use the dataset as is (as it already contains features extracted from the raw data); for our DNNs, we download the raw diffs from GitHub using the following URL <https://www.github.com/{owner}/{repo}/pull/{id}.diff>. The PR description texts and titles are extracted from GHTorrent [57]. For our experiments, we use pull requests from two programming languages: Java and Ruby — due to their differences in syntax and programming style.

¹Gousios et al. also use dataset balancing in their experiments: <https://github.com/gousiosg/pullreqs/blob/master/R/class-mergedecision.R#L90>

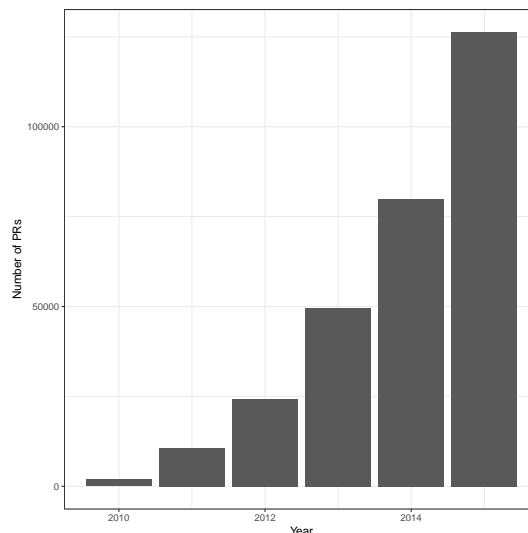


Figure 3.1: Overview of pull requests per year in our data set.

3.1.1. Pre-processing

We tokenize the input using two different procedures. The description and title are tokenized using the default Keras tokenizer². The diffs are tokenized by first lowercasing and then grouping all consecutive alphanumeric characters into tokens. Non alphanumeric characters are seen as one token. For example the following:

```
int foo_bar1 = "test";
```

is tokenized as

```
int, foo, _, bar1, =, ", test, ", ;
```

We use this procedure, instead of a regular programming language parser, since a diff also contains metadata which we incorporate in our research. Furthermore, a PR can contain other sorts of files, such as documentation in the form of Markdown files.

We then generate a vocabulary of the most frequently occurring n tokens exclusively based on pull requests in the training and validation set. We choose $n = 50,000$ for the description and diffs as well as $n = 10,000$ for titles. These values are chosen because increasing them does not yield better results.

This finite set of vocabulary terms also means that we have to deal with out-of-vocabulary terms; we simply remove those, both from the test and the training set. Due to the limited amount of processing power, we only consider the first 150 tokens of the diff and pull request description and the first 20 tokens of the pull request titles respectively. For the case of Java, the average number of tokens is 5.99 (pull request titles), 33.03 (pull request descriptions) and 32,683.20 (diffs) respectively. 95% of these diffs have a token count of 38,000 or less.

3.2. Models

As stated before, our two DNN models differ in their input: Model 1 receives only the raw `diff`, while Model 2 receives the pull request title and description in addition the `diff`.

3.2.1. Model 1

Figure 3.2 shows the model's topology. It consists of an embedding layer followed by an LSTM layer and finally the output layer which is a simple feedforward layer with a single output neuron (using the sigmoid activation function) that outputs the classification (merge/no-merge). We employ an LSTM layer due to its success in natural language processing. LSTMs are well suited to deal with variable

²<https://keras.io/preprocessing/text/#tokenizer>

length input as is the case here. They record an internal state and are thus able to learn long-term dependencies.

The embedding layer converts the one-hot encoded vector input³, into a dense vector of fixed size. This mapping is learned during training. One-hot encoding is the most typical way to represent a word of a vocabulary. Each token of the input is one-hot encoded and presented to the model.

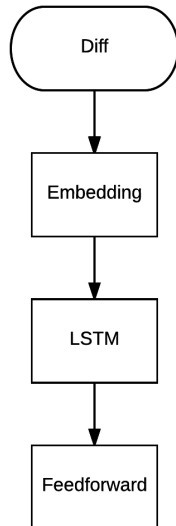


Figure 3.2: Topology of Model 1.

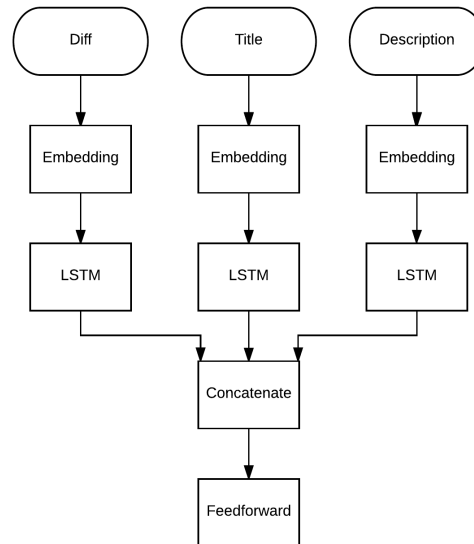


Figure 3.3: Topology of Model 2.

3.2.2. Model 2

In addition to the `diff`, we now also consider the PR title and description, requiring a slight adjustment of our model. We duplicated the topology of Model 1 three times (as shown in Figure 3.3), with each input branch being either the `diff`, the PR title or the PR description. The final output vectors of each LSTM layer are concatenated and then passed to the output layer.

Each of the LSTMs has an auxiliary output with the same loss function as the main output. These loss functions are weighted less than the main output and are used to train these LSTMs individually. This acts as regularization and results in slightly higher accuracy.

3.2.3. Random forest baseline

To train our baseline, we used features that correspond (and were extracted from) the same data that we fed into the DNN models. Specifically, the model we use to train the Random Forest is the following (in R notation):

```

merged ~
  # Project language
  lang +
  # Diff-based features
  files_added_open + files_deleted_open +
  files_changed_open + src_files_open +
  doc_files_open + other_files_open +
  src_churn_open + test_churn_open +
  new_entropy +
  # Description based features
  at_mentions_description +
  description_length +
  
```

³For example, a word in a vocabulary of size 3 can be represented as [0, 0, 1] using one-hot encoding.

feature	description
lang	The programming language the project is mainly written in.
files_added_open	Number of new files added in the PR.
files_deleted_open	Number of deleted files in the PR.
files_changed_open	Number of modified files in the PR.
src_files_open	Number of source code files touched in the PR.
doc_files_open	Number of documentation (markup) files touched.
other_files_open	Number of non-source, non-documentation files touched.
new_entropy	The amount of Shannon entropy the code introduces to the code base.
src_churn_open	Number of lines changed (added + deleted).
test_churn_open	Number of test lines changed.
at_mentions_description	Number of mentions in the description.
description_length	Length of the PR description.
title_length	Length of the PR title.

Table 3.1: Features and their description we use for training the Random Forest baseline

```
# Title based features
title_length
```

We describe each feature we use to train the Random Forest baseline in table 3.1.

3.3. Training

In this section we describe how we train the models .

3.3.1. Model 1 & Model 2

We train both models with RMSprop [28] and use dropout [32] on the LSTM and embedding layers, following good practices in deep learning [58].

Due to the large amount of training time required, we do not perform a grid search through all possible hyperparameter settings, but settle on common ones: for Model 1 and Model 2 we use a learning rate of 0.006, a batch size of 150, an embedding size of 500, an LSTM size of 300 and 20% dropout. In addition, for Model 2 we set the description embedding size to 500 and the title embedding size to 250; the extra loss functions of the diff LSTM, comment LSTM and title LSTM are weighted as follows: 0.3, 0.1 and 0.1 respectively. After training the model on 90% of the training data, we employ the remaining 10% for validation and performed early stopping: training concludes when the validation loss does not improve for 5 epochs. Note, that (slight) changes in those hyperparameters do generally not have an effect on the model's test accuracy.

The training is performed on a workstation featuring 8 cores, 16GB of RAM and an NVIDIA TitanX Pascal GPU with 12GB of RAM. Training Model 2 for 10 epochs takes approximately half an hour on this hardware.

3.3.2. Random forest baseline

To train the Random Forest baseline, we use R's `randomForest` package. We experiment with various settings, specifically the number of trees and the number of features per split. We train our baseline with increasing number of trees and features per split and stop training when the Out Of Box training error stops improving. In both the Java and the Ruby case, this occurred at around 160 trees and 4 features per tree. We therefore use 200 trees and 4 features as the default configuration for all our models. For comparison purposes, we also train a Random Forest (which we call RF-full) using the full set of 40 features, including ones that stem from analyzing developer profiles and the project's source code (recall that these features are derived from data not available to our DNNs), available in the Gousios and Zaidman dataset.

3.4. Experiments

In order to explore different prediction difficulty levels, we run three experiments using the following split strategies:

	Model 1	Model 2	RF	RF-full
Random split				
Java ($n = 37,944$)	0.61	0.63	0.59	0.74
Ruby ($n = 55,004$)	0.60	0.62⁺	0.60	0.72
Time-based split				
Java ($n = 34,767$)	0.55	0.58⁺	0.54	0.73
Ruby ($n = 50,525$)	0.61	0.63⁺	0.60	0.75
Project-based split				
Java ($n = 37,944$)	0.55	0.56	0.57	0.68
Ruby ($n = 55,004$)	0.55	0.57	0.58	0.70

Table 3.2: Test accuracy for the three splits of the data and our two DNN models as well as the Random Forest (RF) baseline. RF-full is the Random Forest trained on the full feature set, instead of only those features derivable from the data the DNN has access to. Model performances that are significantly better than the RF baseline are annotated with a +

- **Random:** We randomly sample training and test cases from the dataset.
- **Time-based:** We order the PRs by submission timestamp, train on the earliest 80% and test on the remaining 20%. As shown in Figure 3.1, 43% of all pull requests were made in 2015, with less than half a percent made in 2010. Our training data stems thus from 2010 to the first half of 2015, while our test data is made up of pull requests from the second half of 2015.
- **Project-based:** We sample 80% of the projects in the dataset for training and consider the remaining projects for testing. This requires the learner to infer general patterns about the input that can be generalized across projects.

3.5. Results

The results of our experiments for the three different splits of the data are shown in Table 3.2, where we report the accuracy on the respective test set. Due to the balanced nature of the data, a random baseline will achieve on average an accuracy of 0.5.

We find that all models outperform the random baseline. Model 1 is inferior to both Model 2 and RF in all cases, which is expected as it is trained using less data than the other two models. Still, even by discarding almost 95% of the diff data, it is able to perform better than the random baseline. The comparison between Model 2 and RF is more interesting, where we see that DNNs outperform RFs in almost all cases and splits. To confirm that the differences are statistically significant, we perform a McNemar χ^2 test between the contingency tables of Model 2 and RF; we find that all differences are statistically significant at $p < 0.01$, except in the case of the Java random split.

The project-based split creates an almost impossible classification task, given the wealth of literature indicating that models in software engineering are not generally transferable among projects (see for example [59]). Indicative is the fact that even the RF-full model performance drops for this task. The performance of both Model 2 and RF is equally suffering.

3.6. Conclusion

In this case study we trained multiple models to predict whether a PR is going to be merged or not. We trained a DNN receiving as input the diff of a PR, a DNN receiving as input the title, description and diff of a PR and a Random Forest using manually engineered features as baseline. The models are trained using three data split strategies: random, time-based and project-based. The DNNs, processing only a small portion of the raw data, performed equally well or better than the Random Forest baseline model in almost all cases. This shows that the raw data contains more information than the researcher and the Random Forest model are able to capture using the engineered features. The performance of the DNN improved when we gave it access to the title and description of a PR, which means that those data sources contain useful information. In traditional machine learning it is possible to find out the importance of a feature by removing it and retraining the model. The more important the feature, the more the performance of the model then drops. Unfortunately, currently it is difficult [60], due to the complexity of DNNs, to extract the features it learned to recognize. However, DNNs can be used to

gain insight into which data source to investigate, as we have shown in this research by incorporating diffs, titles and descriptions as data sources.

4

Finding similar pull requests

On GitHub, most PRs are eventually merged (84.73%). A PR can be closed for various reasons, e.g.: the implementation is not good enough or the PR does not follow the projects conventions. Of all closed PRs, 25% is not merged because another PR has implemented the same or a similar concept [13]. We call those PRs *similar*. The GHTorrent [57] mirror of GitHub contains approximately 29 million PRs. Based on the numbers above, almost 1.1 million pairs of those PRs should be similar.

One of the top challenges integrators of open source projects face, is prioritizing work, when facing multiple concurrent pull requests [61]. Automating the process of finding similar PRs could help integrators prioritize PRs by reducing the noise of multiple PRs implementing the same concept. Additionally, by linking similar PRs we can point contributors to relevant information, such as alternative implementations and earlier discussions about the implemented concept.

We define the problem of finding similar PRs as a binary classification problem: given two PRs, are these PRs similar or not? We consider this task interesting, because it requires domain knowledge of the respective project to solve and we therefore consider it a hard task for humans and for computers. Furthermore, to our knowledge, this is the first study that addresses this problem.

PRs can contain multiple types of files, such as source code and documentation. The source code can be written in a multitude of programming languages. As mentioned in the introduction of this thesis, Hidle et al. [10] show that source code, like natural languages, is likely to be repetitive and predictable. Inspired by this work, we process the information in a PR, including source code, by training models that have shown to perform well on NLP problems. Specifically, we use Word2Vec and Doc2Vec as pretraining techniques and train a siamese convolutional neural network on top of the first and a feedforward neural network and XGBoost on top of the latter. We train these models to predict whether two PRs are similar based on the diffs, the titles and the descriptions of a pair of PRs.

4.1. Data

To be able to train machine learning models we need examples of similar PRs and non-similar PRs. In this section we first explain which methodology we use for collecting similar and non-similar PRs. Then we show how we manually evaluate the quality of the collected data and finally we describe an analysis of the gathered data.

4.1.1. Construction

When a PR is closed the reason is often specified in the corresponding comments. If it is closed because of the existence of a similar PR, this is specified with a reference to said PR. For example:

closing in favor of the duplicate #961

explains that this PR is closed because the PR with identifier 961 is a duplicate of this PR. This syntax is also used for referring to issues instead of PRs, since GitHub models PRs and issues as the same concept internally. We use the existence of these referencing comments to find pairs of similar PRs.

First, we download all comments containing a reference from the GHTorrent [57] database. These comments are matched using the following regex: `/.*#[0-9]+.*/`. Subsequently, we remove the

keyword	count	precision	example
duplicate	4297	0.70	<i>closing in favor of the duplicate #961</i>
dupe	524	0.95	<i>Closing - dupe of #10</i>
dup	321	0.90	<i>This is a dup of #1475</i>
superset	135	0.75	<i>Closing in favor of #1823, as it's a superset of this PR.</i>
continuation	210	0.70	<i>[..] Please see continuation PR #1022</i>
subset	376	0.20	<i>Closing in favor of #4213 (a safe subset of GCAAllocator).</i>
duplicated	751	0.40	<i>Closing this PR and submitted a new one: #2777 This one duplicated a lot of content [..]</i>
resubmission	31	0.95	<i>This is a resubmission of #264 to clean up the commits.</i>
replacement	590	0.60	<i>ok, this is a better replacement for #324</i>
consequence	123	0.00	-
follow-up	415	0.00	-
duplicates	482	0.60	<i>Both this and #79 were closed as duplicates of each other, but it seems that neither was merged.</i>
rehash	11	0.18	<i>Looks like #1208 is a rehash of this. [...]</i>
duplication	434	0.20	<i>:+1: but it seems to me a duplication to #851.</i>
combination	619	0.00	-
favor	18798	1.00	<i>Closing in favor or #5565</i>
part	9982	0.00	-
backport	1232	0.20	<i>Could you review it, please? Thanks! It is backport for #1321</i>
followup	211	0.00	-
favour	4750	0.90	<i>Closing in favour of #1314.</i>

Table 4.1: Potential keywords for searching for similar PRs with for each keyword: the number of comments it occurred in, the precision for retrieving similar PRs based on 20 samples each and an example that refers to a similar PR.

comments that correspond to an issue or refer to an issue instead of a PR, leaving us with PR comments that are referring to other PRs.

Based on the example above, we choose *duplicate* as a potential keyword for finding comments referring to similar PRs. We then use Word2vec to find words used in the same context as *duplicate*. We first preprocess the comments by removing hyperlinks, URLs, directory paths, code, blockquotes and package names, lowercase all comments and then train Word2vec on them for 5 epochs using the Word2vec implementation of Gensim. [62] After training we use cosine similarity to select the 19 most similar words to the word *duplicate* based on the word embeddings generated by Word2vec.

For each of the resulting keywords and the keyword *duplicate* we randomly sample 20 examples and inspect if they refer to a similar PR. The results are shown in Table 4.1 including examples of comments that refer to similar PRs. The keywords *follow-up* and *followup* do refer to other PRs, however, these PRs add functionality on top of the current PR after it is merged. These are therefore not considered similar. We select all comments containing keywords with an accuracy of over 70%, which are: *duplicate, dupe, dup, superset, continuation, resubmission, favor and favour*.

Users refer to similar PRs in comments using the keywords: *duplicate, dupe, dup, superset, continuation, resubmission, favor and favour*.

During the examination of the comments we note that most of the true positives are short comments, while the false positives are relatively long comments. Therefore we exclude comments with more than 4 newlines.

%	file extension
18	js
14	py
13	md
11	rb
11	json
9	php
7	yml
6	txt
5	html
5	java

Table 4.2: Top 10 most occurring file extensions with for each extension the percentage of the PRs it occurs in.

For each matched comment, we extract the PR identifier it is referring to and pair it with the PR the comment corresponds with. We then download all diffs from the following URL <https://www.github.com/{owner}/{repo}/pull/{id}.diff>. The titles and descriptions of the PRs are extracted from GHTorrent.

We count the number of newlines in each diff (first quartile: 22, median: 94, third quartile: 345) and remove all pairs containing at least one diff with more than 28567 lines, which corresponds to the 99th quantile. We also remove all pairs containing at least one diff that contain zero new line characters or the GitHub error page, since those diffs failed to download.

We then split the dataset in a training set (80%), validation set (10%) and test set (10%). In practice, there will be $n(n-1)/2$ pairs of PRs in a repository containing n PRs. To emulate such a real situation, for each similar pair in the datasets we select six random pairs of PRs from the same project and add these as non-similar pairs to the same dataset. We make sure that one of the six non-similar pairs contains PRs with at least one modified file in common. We believe that these examples are more difficult to classify compared to randomly selected PR pairs. If none could be found, we select two random PRs. The number of six non-similar pairs is chosen because of limited hardware resources.

The resulting dataset consists of 154153 pairs of PRs of which 22503 are similar and 131650 are non-similar. These pairs consist of 258991 unique PRs from 9000 projects, which contain a multitude of programming languages. We have counted for each file extension in how many PRs it occurs. Table 4.2 shows the top 10 most occurring file extensions. We note that the PRs do not only contain source code, but also markdown, text, html and configuration files.

For each of the PRs we generate the following representations on which we train our models on:

diff The diff of a PR without the context of the modified lines.

diff+context The diff of a PR including the context of the modified lines.

description+title The concatenation of the title and the description of a PR.

4.1.2. Evaluation

We randomly sample 150 pairs of PRs from the training set, remove the labels and divide them in three. Each set contains 25 similar and 25 non-similar pairs of PRs. The first dataset is manually labeled by the author of this thesis, while the second dataset is split in five and labeled by five raters. The third dataset is used to determine the inter-rater agreement and is labeled by both the first author and the raters (in the same split fashion). A pair of PRs can be labeled as *similar*, *non-similar* or *unknown*.

The first author reports an accuracy of 98%, excluding 1 pair labeled unknown, on the first set, while the group of raters reports an accuracy of 79%, excluding 3 pairs labeled unknowns, on the second set. Combined, this results in an accuracy of 88%.

We calculate the inter-rater agreement by calculating Cohen's kappa [63], which is defined as:

$$\kappa = 1 - \frac{1 - p_o}{1 - p_e} \quad (4.1)$$

Here, p_o is the relative observed agreement among raters and p_e is the chance of agreement. In our case $p_o = 0.83$ and $p_e = 0.5$, resulting in $\kappa = 0.67$, which is called substantial by Landis & Koch [64].

		author	
		similar	non-similar
group of raters	similar	17	1
	non-similar	7	23

Table 4.3: Labels of the group of raters versus the labels of the first author on the third dataset excluding pairs labeled unknown.

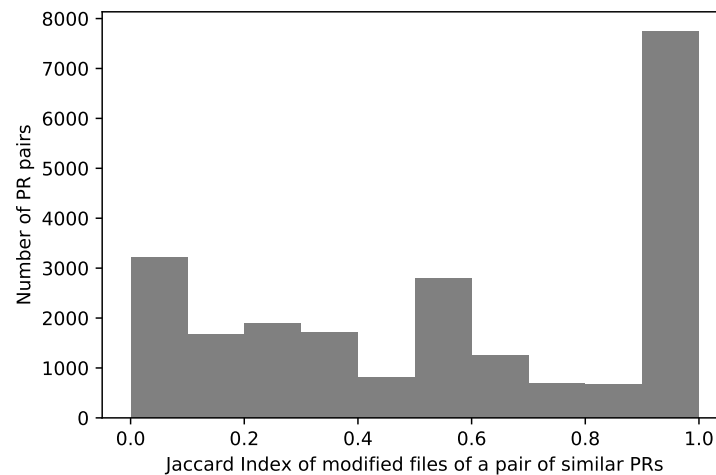


Figure 4.1: Histogram of the Jaccard Index of modified files of similar PR pairs.

Table 4.3 shows how the labels of author compare to the labels of the group of raters on the third dataset. We exclude pairs of PRs that are labeled *unknown*.

4.1.3. Analysis

Because a pair of similar PRs implement similar functionality, we expect such a pair to have modified files in common. To test this hypothesis, we calculate the Jaccard index [65] of the files modified in the collected pairs of similar PRs. The Jaccard index is defined as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \quad (4.2)$$

where A is the modified files in the first PR and B is the modified files in the second PR. The Jaccard index describes to which extent the PRs overlap with respect to the files they modify.

Figure 4.1 shows a histogram of the Jaccard Index of modified files of similar PR pairs. The first quartile, median and third quartile are respectively: 0.23, 0.5 and 1.0. 33% of the pairs have a Jaccard index of 1, meaning that these PRs modify exactly the same files. 8% of the pairs does not contain any overlapping modified files.

33% of similar pairs of PRs modify exactly the same files, on the other hand, 8% of the PR pairs has no overlapping modified files.

It is only useful to submit a PR containing functionality that is not already merged into a project. We therefore expect a PR to be submitted during a short timeframe after the other PR in a similar pair. To investigate this hypothesis, we calculate for each of the collected similar PR pairs the time difference between the submission dates.

Figure 4.2 shows an histogram of the days between submission of PRs in a similar pair. The results confirm our hypothesis: in 50% of the cases the second PR is submitted within 6 days after the submission of the first PR. 75% of the PR pairs are submitted within 36 days (approximately 5 weeks) of each other.

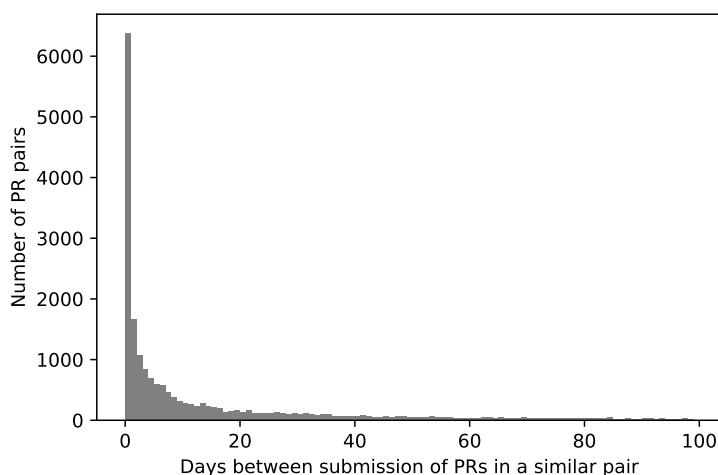


Figure 4.2: Days between submission of PRs in a similar pair.

The time between submission of two PRs in a similar pair is less than 6 days for 50% of the similar pairs. For 75% of the PR pairs, this is less than 36 days (approximately 5 weeks).

4.2. Preprocessing

Neural networks operate on vectors and are thus not able to process source code or text directly. We therefore first tokenize the *diff*, *diff+context* and *description+title*. We then use Doc2Vec and Word2Vec to generate respectively a vector and a sequence of vectors for each of these inputs. Alternatively, we also generate a bag-of-words representation which we use as a simple baseline. In this section we explain each of these preprocessing steps. We use a slightly adapted version of the tokenizing process explained in section 3.1.1

4.2.1. Tokenizing

We indicate an added or removed line with respectively a *LINEADDED* or *LINEREMOVED* token instead of a *+* or *-* symbol, since these symbols appear frequently in code.

We replace the information about the modified file by a *NEWFILE* token, indicating the start of a new file. We do this to prevent the models from making decisions based on the file names. This would prevent a model from being applicable to a project it was not trained on.

Because our dataset contains a large range of different programming languages and even files containing text, implementing a parser per file type would be tedious. Furthermore, this would be slow considering the size of our dataset. We therefore use the same tokenization approach as used in section 3.1.1. Additionally, we split camel cased tokens into multiple tokens.

This procedure produces tokens of reasonable quality without being specifically tailored to a single programming language. It extracts language keywords such as *int* and *if* as a single token and splits type and variable names which consist of multiple words into multiple tokens, identical to [52]. We expect the latter to improve the accuracy of the classifier, since these subwords contain information about the domain and meaning of the piece of code. Apart from source code, this procedure also correctly tokenizes natural language as written in comments and text files. We completely ignore indentation, however, since we remove all whitespace.

4.2.2. Doc2vec & Word2vec

In this case study we use Word2Vec and Doc2Vec instead of the one-hot encoding used in chapter 3. Word2Vec and Doc2Vec can both be trained on unlabeled data. This means that, theoretically, we could train Word2Vec and Doc2Vec on all 29 million PRs submitted on GitHub. By doing this, all tokens

in these PRs (using Word2Vec) and the PRs themselves (using Doc2Vec) are placed in the same vector space. When using one-hot encoding, tokens that did not appear in the labeled trainingset are not taken into account by the neural model, because they do not exist in its vocabulary. Models trained on top of Word2Vec and Doc2Vec representations do not have a vocabulary; they are trained on the learned vector representations of tokens or PRs. They therefore have the potential to generalize to data in the same vector space they are not trained on.

We train a Doc2Vec and Word2Vec model using the same procedure for each of the following input representations: *diff*, *diff+context* and *title+description*. We create a vocabulary based on all tokens that occur at least 5 times in the training and validation set. Based on the recommendations of Lau and Baldwin [66], we train Doc2Vec and Word2Vec together using respectively the PV-DBOW and Skip-gram mode using the Gensim [62] implementation of Word2Vec and Doc2Vec. We train for 10 epochs using an embedding size of 300 and a window size of 5. We start with a learning rate of 0.025 and reduce it 0.002 after every epoch. These parameter choices are based on suggestions by the author of Gensim.¹

Then we generate a vector with dimension 300 for each input using the trained Doc2Vec model. Furthermore we generate a vector with dimension 300 for each token in the input using the trained Word2Vec model, resulting in a sequence of Word2Vec vectors.

We train the Doc2Vec and Word2Vec models on a workstation featuring a Intel(R) Xeon(R) CPU E5-2643 v2 containing 12 CPUs and 128 GB RAM. Training the models on diffs takes approximately 6 hours, whereas training it on descriptions takes approximately half an hour.

4.2.3. Bag-of-words

Bag-of-words is used as a simple and fast baseline for Doc2Vec and Word2Vec, which are relatively complex. We generate a vocabulary based on the training and validation set by selecting all tokens that appear in at least 2 PRs and in less than half of all PRs. We remove all other tokens, because we believe these would add little value, but do increase the training time needed. For each input we then use the count of each token that occurs in the vocabulary as input feature. Note that using this method, we throw away information about the order of the tokens. We expect that this method results in less accuracy, compared to the input representations that take order into account.

4.3. Models

In this section we introduce the five models we use in our experiments. The first uses Doc2Vec to generate vector representations of a pair of PRs. The second uses a siamese convolutional neural network on top of two sequences of Word2Vec embedding to generate a vector representation of a pair of diffs. On top of both representations we use a simple two layer feedforward neural network. The third one is XGBoost on top of Doc2Vec. The fourth and the fifth are linear SVMs on top of bag of words and the Jaccard index of modified files.

We train and test all models for each pair pr1, pr2 two times: one as (pr1, pr2) and (pr2, pr1) to persuade the models to be symmetrically, that is: the models produce the same output independent of the input order. This also applies to the siamese convolutional neural network; The subnetwork in this network is shared and is thus automatically trained symmetrically. The layer on top of the subnetwork, however, is not.

4.3.1. Feedforward neural network

The feedforward neural network we use in this work consists of two layers. The first layer has 2000 neurons and uses the ReLU activation function. We add *dropout* on top of this layer to reduce the chance of overfitting. We use a dropout rate of 0.5. The second layer consists of one neuron and uses sigmoid as activation function. This neuron, due to the use of the sigmoid function, outputs a number between 0 and 1, 0 meaning non-similar and 1 meaning similar. We use two layers instead of one since it improved the performance of the network. Adding a third did not improve performance.

The network receives as input the concatenation of all Doc2Vec vectors representing a pair of PRs. This can be one or a combination of the PR representations described in section 4.1.1.

We train this model using the Adam optimizer with learning rate 0.00007. We keep training until the loss of the model has not improved for 8 epochs and then select the model with the lowest loss on

¹<https://rare-technologies.com/doc2vec-tutorial/>

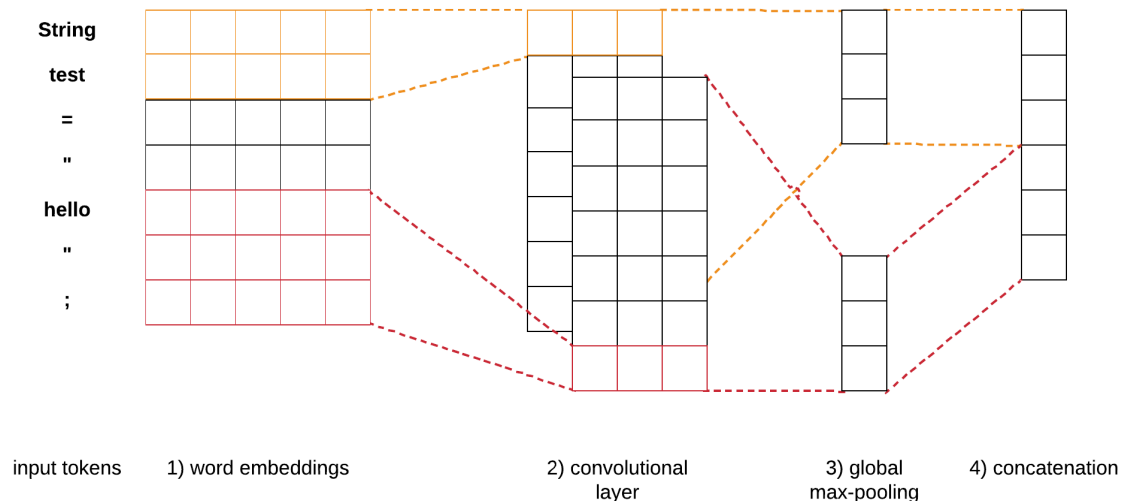


Figure 4.3: Architecture of the convolutional subnetwork based on [22]. For clarity in this figure the word embedding have size 5 and the convolutional layer has filters of length 2 and 3, each containing 3 filters.

the validation set.

4.3.2. Siamese convolutional neural network

The second model we use is a siamese convolutional neural network that uses a CNN as proposed by Kim [22] as subnetwork. We use this model, because it has performed well on various NLP tasks. We do not use an LSTM, because LSTMs process sequences sequentially whereas CNN can process a sequence in parallel and are therefore faster. Each subnetwork accepts as input a sequence of Word2Vec vectors representing a PR. This can be a diff or a title+description. We thus use two identical subnetworks, one for each PR.

The architecture of the subnetwork consists of multiple layers which are displayed in Figure 4.3.

1. Word Embeddings

The subnetwork takes as input a sequence of tokens in the form of Word2Vec embeddings. Tokens that are not in the vocabulary are mapped to a vector containing zeroes.

2. Convolution layer

Subsequently, a convolutional layer applies multiple filters by sliding these over the input.

We use three convolutional layers with filter length 3, 4 and 5 as suggested by Kim [22] with 100 filters each. We use ReLU as activation function.

3. Global max-pooling

Next we apply global max-pooling which selects the maximum value for each filter. We use global max-pooling, since Zhang & Wallace [67] have shown that it performs better than other forms of pooling in the case of this architecture.

4. Concatenation

We concatenate the outputs of the subnetworks. On top of these concatenated vectors we run a feed-forward neural network with the same architecture as described in section 4.3.3.

We train two variants of this model: The first variant is trained end-to-end, called *CNN + Word2Vec*. We only train on diffs in the training set that contain less than 20k tokens because of memory constraints. This accounts for 93% percent of the total dataset. Due to the use of the global max-pooling layer, the features the CNN learns to recognize are translation invariant. The trained network should

therefore generalize to diffs with more than 20k tokens. We test the model on *all* diffs in the test set, but cut them off at 20k tokens.

The training of the second variant, called *CNN + Word2Vec + Euc*, consists of two steps. We first train the convolutional subnetwork without the feedforward neural network on top of it by minimizing/maximizing the euclidean distance between the output vectors of the two subnetworks. We use contrastive loss as used by Hadsell, Chopra and LeCun [68]. We again only train on diffs in the training set that contain less than 20k tokens

Subsequently we cut off *all* diffs at 20k tokens and feed them through the convolutional network. This results in a vector representation of each diff. We then train the feedforward neural network on top of all pairs of diffs in the training set. Afterwards we concatenate both models and thus create an end-to-end siamese convolutional network. We test the model on *all* diffs in the test set, but cut them off at 20k tokens.

Both models are trained using the Adam optimizer with a learning rate of 0.0001. We keep training until the loss of the model has not improved for 8 epochs and then select the model with the lowest loss.

4.3.3. XGBoost

We use XGBoost [69] on top of the Doc2Vec representation. XGBoost is used since it is considered a high performance traditional machine learning algorithm, as shown by multiple winning models in Kaggle competitions [70].²³⁴ This model receives the same input as described in section

We train this model by adding new decision trees until the AUC has not improved on the validation set for 100 added trees. We then test the model on the best performing model based on the AUC on the validation set.

4.3.4. SVM

We use a linear SVM on top of the bag of words representation as a baseline for the more complex Word2Vec and Doc2Vec representations. A linear SVM with bag of words is used because it is a simple and standard baseline [67] for NLP problems. Since Word2Vec and Doc2Vec have already shown to perform well for NLP problems, we only use this baseline on top of on diffs. Furthermore, we use a linear SVM on top of the Jaccard index as another baseline, because we suspect that the overlap of modified files is an important feature in detecting similar PRs.

4.4. Experiments

We conduct five experiments to investigate the performance of the five models in different settings. In experiment 1, 2 and 3 we only train on the *diff* representation on a balanced dataset to investigate the performance of the on NLP inspired models on source code. In experiment 4 and 5 we also include the *diff+context* and *title+description* representations and investigate the performance of these models on an unbalanced dataset. This resembles a practical setting more closely. In this section we explain the set up of each of the experiments.

4.4.1. Experiment 1

To get an idea of the performance of each of the models, we first train the models on a smaller, balanced, dataset. This dataset only contains the similar pairs of PRs and the non-similar pairs of PRs containing at least one overlapping modified file.

4.4.2. Experiment 2

With the second experiment we investigate the performance of the models on projects they are not trained on. We do not include the SVM + BoW model because of the bad performance in the previous experiment. We first select the twenty projects with the most pairs of PRs in our balanced dataset. We exclude project *pyupio/demo* from this list, since the project barely has content and contains only PRs submitted by a bot. This leaves us with the twenty projects shown in Table 4.4. We place the PRs

²<http://blog.kaggle.com/2015/12/03/dato-winners-interview-1st-place-mad-professors/>

³<http://blog.kaggle.com/2015/11/30/flavour-of-physics-technical-write-up-1st-place-go-polar-bears/>

⁴<http://blog.kaggle.com/2015/08/26/avito-winners-interview-1st-place-owen-zhang/>

project	count
symfony/symfony	630
Homebrew/homebrew	318
bitcoin/bitcoin	298
rails/rails	258
edx/edx-platform	254
avocado-framework/avocado	246
joomla/joomla-cms	226
docker/docker	220
angular/angular.js	214
KSP-CKAN/NetKAN	200
kubernetes/kubernetes	194
apache/spark	190
RIOT-OS/RIOT	174
TryGhost/Ghost	152
nodejs/node	144
rust-lang/rust	138
symfony/symfony-docs	138
ManageIQ/manageiq	134
puppetlabs/puppet	126
pydata/pandas	124

Table 4.4: Top 20 project with the most PR pairs in our dataset, excluding pyupio/demo.

corresponding to these projects, accounting for approximately 10% of the data, in the test set and split the remaining PRs randomly between the validation set and training set, where the validation set and training set contains respectively approximately 10% and 80% of the total dataset.

4.4.3. Experiment 3

In experiment 3 we investigate whether we can improve the results of experiment 2 by training Word2Vec and Doc2Vec on arbitrary and unlabeled PRs from projects in the test set. The intuition behind this is that by including these PRs during training, Word2Vec and Doc2Vec will place the tokens specific to the domain of these projects in the same vector space as all tokens from the training- and validation set. This approach is useful in practice, because there may not be enough labeled data available to train a model for a specific project, whereas most open source project contain an abundance of arbitrary PRs to train on.

To test this hypothesis we download 200 random PRs from each project in the test set. We exclude the PRs that are already in the test set. We then train Doc2Vec and Word2Vec on these PRs and the existing training- and validation set. Finally, we retrain all models used in the previous experiment on top of the new Word2Vec and Doc2Vec representations.

4.4.4. Experiment 4

In the fourth experiment we exclude the siamese CNN models based on the fact that these require an order of magnitude more memory and training time compared to the Doc2Vec models. We train the *FFNN + Doc2Vec* and *XGBoost + Doc2Vec* models on the full unbalanced dataset using the following combinations of input representations:

- **diff**
- **diff+context**
- **title+description**
- **diff & title+description**
- **diff+context & title+description**

model	validation accuracy	test accuracy
SVM + Jaccard Index	0.656	0.651
SVM + BoW	0.554	0.552
CNN + Word2Vec	0.693	0.698
CNN + Word2Vec + Euc	0.749	0.753
FFNN + Doc2Vec	0.782	0.765
XGBoost + Doc2Vec	0.783	0.771

Table 4.5: Accuracy of each of the models on the validation set and test set trained on diffs in experiment 1.

To prevent the models from learning the majority case (non-similar), we scale the loss for the similar PR pairs by a factor n_count/p_count where n_count is the number of non-similar samples in the training set and p_count is the number of similar samples in the training set.

4.4.5. Experiment 5

As described in section 4.1.3 75% of the PRs in a similar PR pairs is submitted within a timespan of approximately 5 weeks. We can use this finding to reduce the search space for finding similar PRs. Instead of checking all $n \cdot (n - 1)/2$ pairs in a project, we only have to compare pairs of PRs that have been submitted with a timespan of x . Based on earlier research on GitHub [13, 71] we choose $x = 90$ days (3 months) which includes approximately 85% of the similar PR pairs.

To test the classifiers on such a scenario, we collect a new dataset of non-similar PRs. For each project in the training, validation and test set, we divide the PRs in windows of 90 days. We remove windows containing less than 2 PRs. Then we count the number of similar PR pairs n per project and collect $6 \cdot n$ pairs of non-similar PRs from that project by first selecting a window randomly and then selecting two PRs randomly from that window. Finally, we remove PR pairs that contain two identical PR ids. This results in 115310 pairs of non-similar PRs, which is 16340 pairs less than the dataset used in experiment 2. In the test set 83.8% of the pairs is non-similar, whereas in the previous dataset this was 85.4%. We report the performance of our models in two settings: *setting 1* in which we test the models as done in the previous experiments and *setting 2* in which we classify each pair of similar PRs that has been submitted within a timespan of more than 90 days as non-similar, since in practice using the described procedure we would not have been able to find these pairs.

4.5. Results

In this section we describe the results of the five experiments.

4.5.1. Experiment 1

The results of experiment 1 are shown in table 4.5. We report the accuracy on the validation and test set. The *SVM + BoW* does only improve slightly over the random baseline (having an accuracy of 50%) which signals that this model is not a good fit for this problem. All proposed models perform better than both the *SVM + BoW* and *SVM + Jaccard Index* baselines. The siamese CNN pretrained model (CNN + Word2Vec + Euc) performs better than the vanilla siamese CNN (CNN + Word2Vec), which tells us that pretraining siamese CNNs is beneficial. However, it performs worse than the two Doc2Vec models. The XGBoost model outperforms the FFNN model only slightly.

4.5.2. Experiment 2

The results of experiment 2 are shown in table 4.6. The test accuracy of all models does not decrease compared to experiment 1. We expect that the overlap in domain of projects between the test- and trainingset explains why the accuracy does not decrease. For example: *symfony/symfony* and *rails/rails*, both webframeworks, are in the testset, whereas *cakephp/cakephp* and *laravel/laravel* are webframeworks in the trainingset. The higher test accuracy compared to the validation accuracy of the *SVM + Jaccard Index* model and the higher accuracy compared to the results of the previous experiment suggest that in this testset overlapping files and similarity are more correlated than in the test set of the previous experiment. Thus we can interpret this test set as being slightly easier. This could explain the increase of the test accuracy of all other models.

model	validation accuracy	test accuracy
SVM + Jaccard Index	0.662	0.690
CNN + Word2Vec + Euc	0.719	0.743
FFNN + Doc2Vec	0.753	0.781
XGBoost + Doc2Vec	0.766	0.794

Table 4.6: Accuracy of each of the models on the validation set and test set trained on diffs in experiment 2.

model	validation accuracy	test accuracy
CNN + Word2Vec + Euc	0.725	0.741
FFNN + Doc2Vec	0.764	0.796
XGBoost + Doc2Vec	0.761	0.795

Table 4.7: Accuracy of each of the models trained in experiment 3 on the diffs on the validation- and test set.

4.5.3. Experiment 3

The results of experiment 3 are shown in Table 4.7. Compared to experiment 2, the accuracy of *CNN + Word2Vec + Euc* on the test set decreases slightly while it increases on the validation set. The accuracy of *XGBoost + Doc2Vec* on the test set increases while the accuracy on the validation set decreases. The accuracy of *FFNN + Doc2Vec* increases for both the validation and test set. Based on these conflicting results, we cannot conclude that adding extra unlabeled data is beneficial.

4.5.4. Experiment 4

We report the precision, recall, AUC and F1-score of each model on the test set in table 4.8. We note that the *SVM + Jaccard Index* baseline only produces a decision instead of a probability. Therefore the AUC of that model is relatively low compared to all other models. All models perform better than the *SVM + Jaccard Index* baseline on all metrics. The models that use *diff+context* report a higher recall, but lower precision, compared to using *diff*. Combining the title, description and diff results in a better performing model than using these data sources individually. The FFNN models have a higher precision, but lower recall, compared to the XGBoost models.

4.5.5. Experiment 5

Table 4.9 shows the precision, recall, AUC and F1-score of each model tested in *setting 1*. Compared to experiment 4 the precision of the best models increases, which can be explained by the reduced number of non-similar PR pairs vs similar PR pairs. The recall of all models, on the other hand, drops by 10 percent point. All performance metrics of the *SVM + Jaccard Index* baseline increase, probably due to not explicitly selecting non-similar PRs with overlapping files. However, all models still perform better on all metrics compared to this baseline, with two exceptions: the recall of *FFNN + title+description* and the precision of *XGBoost + title+description*.

Table 4.10 shows the precision, recall, AUC and F1-score of each model tested in *setting 2*. Com-

model	input	precision	recall	AUC	F1 score
FFNN + Doc2Vec	diff	0.659	0.683	0.888	0.671
	diff+context	0.610	0.722	0.898	0.661
	title+description	0.577	0.809	0.926	0.674
	diff & title+description	0.690	0.795	0.951	0.739
	diff+context & title+description	0.657	0.823	0.951	0.731
XGBoost + Doc2Vec	diff	0.569	0.743	0.891	0.645
	diff+context	0.549	0.758	0.898	0.637
	title+description	0.533	0.840	0.919	0.652
	diff & title+description	0.629	0.846	0.953	0.722
	diff+context & title+description	0.627	0.847	0.952	0.720
SVM	Jaccard Index	0.451	0.609	0.741	0.519

Table 4.8: Precision, recall, AUC and F1 score of each input representation combination and trained model in experiment 2.

model	input	precision	recall	AUC	F1 score
FFNN + Doc2Vec	diff	0.617	0.710	0.888	0.661
	diff+context	0.643	0.702	0.902	0.671
	title+description	0.546	0.616	0.850	0.579
	diff & title+description	0.711	0.696	0.916	0.703
	diff+context & title+description	0.717	0.698	0.922	0.707
XGBoost + Doc2Vec	diff	0.611	0.751	0.898	0.674
	diff+context	0.585	0.767	0.902	0.664
	title+description	0.477	0.665	0.842	0.555
	diff & title+description	0.663	0.769	0.923	0.712
	diff+context & title+description	0.669	0.766	0.924	0.715
SVM	Jaccard Index	0.514	0.686	0.780	0.588

Table 4.9: Precision, recall, AUC and F1 score of each input representation combination and trained model in experiment 3 setting 1.

model	input	precision	recall	AUC	F1 score
FFNN + Doc2Vec	diff	0.580	0.609	0.763	0.594
	diff+context	0.609	0.607	0.777	0.608
	title+description	0.511	0.536	0.733	0.523
	diff & title+description	0.679	0.599	0.788	0.636
	diff+context & title+description	0.687	0.605	0.795	0.643
XGBoost + Doc2Vec	diff	0.574	0.645	0.772	0.607
	diff+context	0.549	0.662	0.776	0.600
	title+description	0.442	0.577	0.727	0.500
	diff & title+description	0.629	0.663	0.795	0.646
	diff+context & title+description	0.637	0.665	0.797	0.651
SVM	Jaccard Index	0.481	0.600	0.737	0.533

Table 4.10: Precision, recall, AUC and F1 score of each input representation combination and trained model in experiment 3 setting 2.

pared to *setting 1*, the recall of each model drops by approximately 10 percent point, which is caused by classifying 15% of the similar PR pairs as non-similar without showing these to the models, due to containing PRs that have been submitted within a timespan of more than 90 days. The precision of each model drops by 3-4 percent point which is caused by less similar pairs being classified as similar and thus the relative number of correctly classified similar pairs vs incorrectly classified non-similar pairs decreases.

4.6. What did the models learn

We attempt to explain the *FFNN + Doc2Vec* model trained on *title+description* in section 4.4.5 using LIME [72], a tool that attempts to explain blackbox models by creating a linear model around a test example. We use the model trained on *title+description*, since titles and descriptions are in general shorter than diffs and therefore easier to visualize. We analyze each pair of PRs by keeping the input from one of the PRs constant, while varying the input from the other PR. We do this twice to visualize the importance of words in both PR descriptions. All examples have been taken from the test set, which means that the model has not seen these examples before.

We show visualizations in figure 4.4 of two pairs of PRs that are not similar and which the model classifies as non-similar with high confidence. It is interesting to note that the model is able to classify the first pair of PRs ⁵ as non-similar, while both explain the addition of tests. It clearly identifies the word *test* as important as shown by the orange highlight, but based on the words *host* and *header* in the first PR and the words *errors* and *logging* in the other PR it decides these are not similar. The visualization of the second pair of PRs ⁶ in figure 4.4 shows again that the model correctly differentiates

⁵<https://github.com/FriendsOfSymfony/FOSHttpCache/pull/2>, <https://github.com/FriendsOfSymfony/FOSHttpCache/pull/6>

⁶<https://github.com/thoughtbot/ember-cli-rails/pull/452>, <https://github.com/thoughtbot/>

between two PRs based on words that describe the functionality being modified; *render* in the first PR and *deployments*, *directories* and *capistrano*, which is a tool for deployment, in the second PR. What is interesting is that the model highlights the number 451, which is a reference to an issue, as signaling similarity. This could mean that the model pays attention to references to issues or that the model has learned to recognize an irrelevant feature.

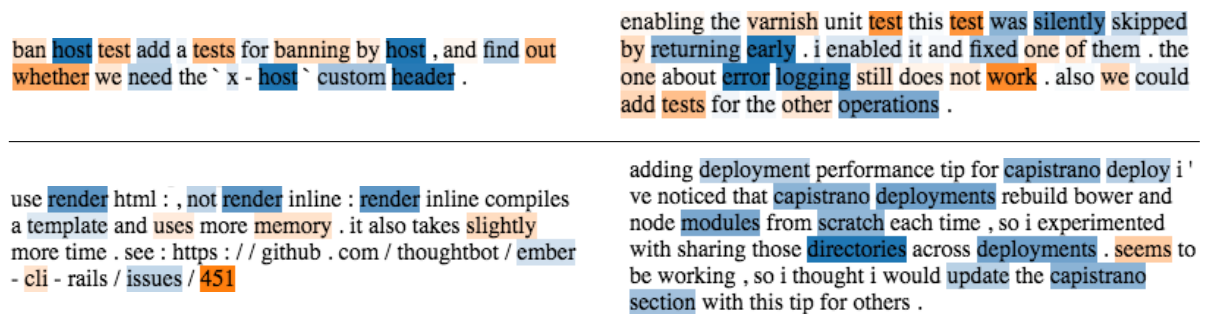


Figure 4.4: Visualization of the title+description of two pairs of PRs that the model correctly classifies as non-similar. Blue highlighted words signal non-similarity, while orange highlighted words signal similarity. Less opaque means more importance.

In figure 4.5 we show an example ⁷ that the model mistakenly classifies as non-similar, but with low confidence. While both descriptions are similar, they describe general git related functionality, which the model clearly struggles with. It could be the case that these words are often used in PRs and are therefore not very distinctive.

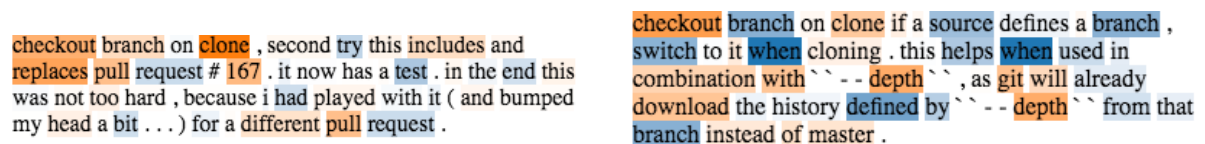


Figure 4.5: Visualization of the title+description of a pair of PRs that the model incorrectly classifies as non-similar with low confidence. Blue highlighted words signal non-similarity, while orange highlighted words signal similarity. Less opaque means more importance.

In figure 4.6 we show two examples that the model correctly classifies as similar with high confidence. In the first example ⁸ the model not only highlights words that are similar to *coordinates*, such as *seconds*, *coordinates*, *angle*, *deg* in both descriptions, but also words that are similar to errors such as *exceptions* and *warnings*. It even takes into account the code included in this description.

In the second example ⁹ the model finds *route*, *state*, *optional* and *parameters* important in both descriptions. It marks *parsed* as signaling non-similarity in the first PR, since the second PR does not mention parsing.

Finally, we show in figure 4.7 a non-similar example ¹⁰ that the model incorrectly classifies as similar with low confidence. Both PRs describe *window* and *screen*, which make them look similar. What is interesting is that the model notes that the word *animation* in the second description signals non-similarity, while it has a meaning similar to *draw* in the first description. It however does so with small weight. In both descriptions it highlights irrelevant words as signaling non-similarity, such as *names*, *if*, *lack*, *often*, *any*, *in*.

These examples are all selected based on explainability and are thus only anecdotal evidence. These are by no means enough to fully explain or trust the model. However, based on this exploratory research, we gain some insight in how the model operates. Furthermore, this tool could be used to provide explainability to users of the model.

ember-cli-rails/pull/459
⁷<https://github.com/fschulze/mr.developer/pull/169>, <https://github.com/fschulze/mr.developer/pull/167>
⁸<https://github.com/astropy/astropy/pull/990>, <https://github.com/astropy/astropy/pull/947>
⁹<https://github.com/aspnet/Routing/pull/209>, <https://github.com/aspnet/Routing/pull/208>
¹⁰<https://github.com/Hammerspoon/hammerspoon/pull/300>, <https://github.com/Hammerspoon/hammerspoon/pull/354>

parse minutes or seconds equal to 60 with warning out - of - bounds angles with seconds == 60 or minutes == 60 result in warnings instead of exceptions . many data sets , including the fermi gbm grb catalog , give sexagesimal angles that are improperly rounded . example :

```

!!!
icrscoordinates ('00 : 60 00 : 60 ' , unit = (' deg ' , ' deg ' ))
warning : an invalid value for ' minute ' was found (' 60 ' ) ;
must be in the range [ 0 , 60 ) . [ astropy . coordinates . angle
_ utilities ] | icrscoordinates ra = 1 . 00000 deg , dec = 1
. 00000 deg | some new test cases are included . this addresses
# 947 , without introducing any new arguments .

```

add relax = keyword argument to angle constructors if set to `true` , then out - of - bounds angles (such as seconds | = 60 , as in `00 : 20 : 60 . 0 `) result in warnings instead of exceptions . many data sets , including the fermi gbm grb catalog , give sexagesimal angles that are improperly rounded . the default is `relax = false` ; if `relax = ` is not specified , then out - of - bounds angles continue to produce exceptions . example with default behavior :

```

!!!
icrscoordinates ('00 : 60 00 : 60 ' , unit = (' deg ' , ' deg ' ))
error : illegal minute error [ astropy . coordinates . angle
_ utilities ] traceback ( most recent call last ) : etc . example
with ' relaxed ' behavior : !!! icrscoordinates ('00 : 60 00 :
60 ' , unit = (' deg ' , ' deg ' ) , relax = true ) warning : [
astropy . coordinates . angle _ utilities ] | icrscoordinates ra =
1 . 00000 deg , dec = 1 . 00000 deg |

```

expose parsed route so more detailed route is available (i . e . optional state) this adds a property to `template route` which allows access to the `parsed template` state of the target route . will be used in glimpse to indicate to the user what parameters are optional vs not , etc

add access to optional state of parameters to template route this adds a property to `template route` which allows access to the optional state of parameters of the target route . will be used in glimpse to indicate to the user what parameters are optional vs not .

Figure 4.6: Visualization of the title+description of two pairs of PRs that the model correctly classifies as similar with high confidence. Blue highlighted words signal non-similarity, while orange highlighted words signal similarity. Less opaque means more importance.

add hs . tabs module this module allows a certain kind of tabbing of the windows of any osx app . i originally wrote it to tab emacs frames but it might also be useful for other apps like finder that don ' t have tabs but should . basically it organizes windows so that they are always all in the same position by listening to events . it then uses the drawing module to draw somewhat - nice looking tabs in the title bar with the window names when any window in the stack is focused . it also exposes functions that allow for binding keys to switch between tabs and enumerate the tabbed windows and suchlike . it currently does not support clicking on the tab names since `hs . drawing` has no click event support . ! [screen shot 2015 - 04 - 19 at 5 51 11 pm] (https : // cloud . githubusercontent . com / assets / 887610 / 7826551 / 660de62c - 03e9 - 11e5 - 8544 - ad861f6aa5e3 . png) i ' ve been using this module as a core part of my daily workflow for multiple weeks now so i ' m fairly sure it holds up well . i think i ' ve worked out all the bugs concerning apps crashing and restarting and so on .

fix chained actions on hs . window workaround for issues with chained actions when hs . window . animation duration | 0 . ideally the (ugly) hs . timer should be removed , and the corresponding cleanup (`init . lua` row 98) should instead happen in a function that in turn gets called from animation did stop / end in `internal . m` . i don ' t know if that ' s possible ; if it is , i surely have no idea how . example for testing :

```

``` lua local w = hs . window . focused window
() w : move to screen (w : screen () : next ()) : move to
screen (w : screen () : previous ()) : move to unit ({ x = 0
, y = 0 , w = 0 . 5 , h = 0 . 5 }) `` the ` move to screen ` s
should cancel out and the window should just move to the
top left quadrant of its current screen , if not already there . -
- - - the changes seem potentially dangerous wrt gc (see
comment) * and * lack thereof if the weak tables don ' t
work as i expect (i don ' t use them very often) , so close
scrutiny / testing is warranted . (@ cmsj suggested on irc as
a possible alternative to strongly advise , in the
documentation , to set ` animation duration ` to 0 if any
chaining might occur)

```

Figure 4.7: Visualization of the title+description of a pair of PRs that the model incorrectly classifies as similar with low confidence. Blue highlighted words signal non-similarity, while orange highlighted words signal similarity. Less opaque means more importance.

## 4.7. Threats to validity

**Internal validity** In this case study we used Doc2Vec to map documents to vectors. The document to vector mapping of a trained Doc2Vec model is non deterministic and therefore outputs a slightly different vector each time it is given the same input. Because of limited hardware resources, we preprocessed all inputs by mapping them to vectors using a trained Doc2Vec model, before training and testing all models on top of this data. The models therefore received the same Doc2Vec vectors each time they were trained on a certain pair of PRs. The models trained on Doc2Vec vectors, as is, could therefore be slightly unstable when used in practice. However, we believe that by training the models on Doc2Vec vectors that are newly generated each epoch, the models will become more robust to this. We believe that this noise could even act as regularization, similar to Dropout, and therefore result in slightly better performing models.

**External validity** In this work we used a similar pair to non-similar pair ratio of approximately 1:5,

again due to hardware limitations. Symfony<sup>11</sup>, a large project on GitHub, received 633 PRs between 11 Juli 2017 and 10 September 2017. The similar pair to non-similar pair ratio will therefore be lower in practice. When a new PR is submitted and one wants to check whether a similar PR exists, it has to be compared to, in this case, 633 other PRs. As a consequence, we expect the precision of the models trained in this work to be lower in practice.

## 4.8. Conclusion

In this case study we used the unsupervised deep learning technique Word2Vec to collect a dataset of similar PRs with high accuracy. The PRs in this dataset modify various file types among which numerous programming languages. In the case of 85% of the similar PR pairs, the PRs are submitted within a 90 days timespan. We randomly selected PR pairs that have been submitted within a 90 days timespan from the projects in the collected dataset containing similar PRs and use these as non-similar pairs. This resulted in a unbalanced dataset with a non-similar pair vs similar pair ratio of 5.17:1. We trained multiple models on top of the Word2Vec representation of a diff and combinations of the Doc2Vec representations of the diff, title and description of a PR to determine whether a pair of PRs is similar or not. We show that these models perform better than the baselines, one using a bag-of-words representation of diffs and one using the Jaccard Index of modified files in a pair of PRs as feature and we show that the proposed models generalize to projects they are not trained on. The best performing models are able to classify pairs of PRs as similar with a precision and recall of respectively 68.7% and 60.5% and 63.7% and 66.5% in this dataset. These models are a feedforward neural network and a XGBoost model on top of the Doc2Vec representations of the concatenation of the title and description of a PR and the Doc2Vec representation of a diff including the lines describing the context of the modified lines.

---

<sup>11</sup><https://github.com/symfony/symfony/>



# 5

## Lessons learned

In this thesis we have shown that neural networks are powerful models that are able to learn from raw data. Unfortunately, there is no such thing as a free lunch. In this section we explain some of the disadvantages of neural models we discovered during this thesis and we think one has to keep in mind when considering the use of neural models. We first explain that training neural models are costly. Then we explain why optimizing neural models and explaining what a neural model learned is difficult.

### 5.1. The cost of training neural models

Neural models are powerful tools that are able to learn complex functions. Nonetheless, this power comes at a cost: neural models have a huge number of parameters that have to be trained. Furthermore, due to the large number of parameters, large datasets are needed to train these models successfully without them overfitting. Training neural models thus requires lots of computation power; it can take hours, days or even weeks. Using a GPU instead of a CPU can speed up training neural models massively, up to more than ten times faster, due to being able to perform many linear algebra computations in parallel. However, even when using GPUs, training neural models is costly. Jiang et al. [53], for example train their models for 38 hours using a Nvidia GeForce GTX 1070. Fu & Menzies [73] show that using an SVM, they are able to achieve similar and sometimes better results than a CNN on predicting whether two questions posted on Stack Overflow are semantically linkable. Training the SVM takes 10 minutes, whereas training the CNN takes 14 hours. Fu & Menzies thus urge researchers to be critical about the use of expensive models and to always include a simple, less costly, baseline.

### 5.2. Optimizing neural models

Implementing a neural model can be done in a few lines of code using one of the available deep learning libraries. Optimizing a neural model, on the other hand, is difficult due to the enormous amount of hyperparameters one has to choose and the cost of training a neural model. It is therefore often called a black art.

First one has to choose an architecture of which the most common are LSTMs, CNNs or regular FFNNs. These architectures can even be combined to create more complex architectures, such as an Encoder-Decoder LSTM, as used in machine translation, or a CNN+LSTM which is used in recognizing objects in videos. Furthermore techniques such as *attention* [74] and *skip connections* [7] can be used. Then one has to choose the number of layers, the activation function of each neuron, the number of neurons in each layer and if and where to use Dropout. Subsequently, the optimizer, for example Adam or RMSprop, has to be chosen which can have multiple tuneable parameters, among which the learning rate. Finally one has to decide when to stop training the model, which can be done using *early stopping* or by limiting the number of training epochs. This is only a small summary of all parameters that can be tuned, which can potentially have an effect on each other.

Hyperparameter tuning is often done using a combination of experience and automatic optimization, due to the lack of theory about how neural models are able to learn. A researcher once compared deep

learning to “building bridges without having knowledge about physics”<sup>1</sup>. A lot of current knowledge and improvements are based on empirical research on small variations of best practices, instead of understanding how and why certain best practices work. It is therefore important to keep up with the current state of the art and best practices. Most high-level deep learning libraries implement some of these best practices by default, such as the recommended initialization technique per architecture type. Furthermore grid search should not be used for hyperparameter optimization. Random search [75] and Bayesian optimization [76] produce models that are as good or better than models optimized using grid search within a fraction of the computation time. Most neural models are general purpose models. We, therefore, recommend repository mining researchers to first take into account state of the art neural models in other domains, such as NLP, when selecting an architecture, before trying a novel architecture.

### 5.3. Explaining neural models

In this thesis we have trained multiple neural models and have shown their performance, but we have never explained what the models actually learned. This, however, is an important part of using machine learning. There are three main reasons for the need to explain a model:

First, based on performance metrics alone, one can never be sure that the model has successfully learned to do the task it was trained for. Yudkowsky [77] gives a good example of how a model can fail to learn the right features, while performing very well on the test set: researchers trained a model to recognize camouflaged tanks in a forest and based on the accuracy on the test set, the model was working perfectly. However, when trying it out in practice, the researchers found out that the model did not recognize tanks at all. It turned out that the pictures with tanks had been taken on cloudy days, while the pictures without tanks had been taken on sunny days. The model thus learned to recognize cloudy and sunny skies instead of tanks. This flaw could have been spotted if the researchers were able to see what the model actually learned.

Secondly, the user should be able to trust the model. Imagine a model that is able to recognize a disease with high accuracy, but only outputs whether the patient has that disease or not. Based on this output alone, it is hard for the doctor and the patient to be able to trust the model. However, if the model explains how it came to that decision, it can be verified by the doctor and this therefore results in the doctor having more faith in the model. In the case of finding similar PRs, the model could highlight the words in a PR description or the tokens in a diff that it finds important for its decision. In the case of merge prediction, the model could explain to the submitter and the integrator why the PR should not be merged. The integrator can then verify the correctness of this decision and the submitter can resolve the issues with the PR.

Thirdly, it can be used to learn more about the data. In the case of merge prediction, it would be interesting to learn what the model learned to recognize, so we can learn what the most common reasons are for a PR to be refused. We can then use this information to educate GitHub users.

Neural models are extremely hard to explain due to their complexity. In computer vision, multiple tools have been developed which are able to explain what a neural model learned. These range from explaining a decision [72] to visualizing the features the model learned to recognize [78]. The development of tools to explain models in NLP, however, is still emerging, but some techniques have already been developed.

Arras et al. [79] and Winkler & Vogelsang [80] use a similar approach: they use what Arras et al. call layer-wise relevance propagation (LRP) to identify which words are important for a certain classification of a CNN.

Li et al. [81] take another approach: they use a back-propagation strategy to determine which words have the most impact on the final result. Furthermore, they plot the activations of neurons to visualize how they react to words.

Kádár et al. [82] use a simple method: they remove a word from the sentence and compare the output of the model on the remaining words with the output of the model on the full sentence. The greater the difference between those outputs, the more important the word is.

Riberio et al. [72] have developed the toolkit LIME, which is able to explain which words are important for the decisions of a model. They do this by varying the input to the model and then learning a linear model around it.

<sup>1</sup><https://www.technologyreview.com/s/608911/is-ai-riding-a-one-trick-pony/>

All the described techniques have in common that they try to measure the effect of individual words on the decision made by the model. In section 4.6 we have done an attempt to explain a neural network using LIME. In this task, this technique delivered sensible results, but imagine a model that predicts whether a PR should be merged or not and as part of that task has learned to recognize valid and invalid lines of Java code. In such an example, not the individual tokens are important, but each line as a whole. When perturbing a valid line of Java code, the line of code will become invalid and therefore each token in the line will be important for the decision of the model. When for example a semicolon is missing and therefore the line is invalid, none of the tokens in the line is important for the decision of the model. Only the missing semicolon is. Based on the techniques described above, which all investigate the effect of each token on the output of the model, one is not able to recognize this complex feature the model learned to recognize. Recognizing complex features learned by neural models is therefore still extremely difficult and requires lots of effort as shown by Karpathy et al. [60] and Radford et al. [83] who analyzed the behavior of an LSTM by inspecting the activations of individual neurons.





# 6

## Future work

In this section we explain some of our ideas for future work. We first explain what improvements can be made to the case studies in this work. Then, we describe two ideas for new research topic, based on promising results from related work.

### 6.1. Predicting pull request merge decisions

In this work we have limited the number of tokens due to LSTMs being very computational intensive. In the second case study, we have shown that CNNs, using Word2Vec, and other models, using Doc2Vec, are able to process diffs more efficiently and produce sensible results. It would therefore be interesting to investigate the performance of these models on the PR merge prediction task using full diffs, instead of only the first 150 tokens.

We have investigated the trained model using three split strategies to get an idea about how well the model generalizes. However, the model is still a black box in the sense that we do not know what the model learned to recognize. Using a tool such as LIME, it is possible to investigate which parts of a diff the model takes into account when making a prediction.

### 6.2. Finding similar pull requests

In the second case study, we have trained multiple models to detect similar PRs. We have trained these models on PRs containing a range of different file types. To gain more insight in the power of these models, it would be interesting to investigate whether these models are able to generalize to file types or programming languages they were not trained on.

### 6.3. New research ideas

As described in chapter 2, Huo et al. [51] use CNNs to match bug reports with relevant files. GitHub also support reporting issues, but these issues are not only used for bug reports. They are also used for feature requests and even questions about the project. Most projects have a large amount of concurrent open issues. For example, Symfony<sup>1</sup> currently has 667 open issues, whereas Keras<sup>2</sup> has 850 open issues. By automatically listing relevant files, these issues can be solved faster. In the case of a bug report or a feature request, a user could instantly see which files should be modified to solve that issue. In the case of a question, the user can be pointed out where to look for the answer.

In GitHub, commits can automatically close issues by using specific keywords<sup>3</sup> and a reference to the specific issue in the commit description, such as *closes #123*. It is possible to create a dataset of <issue, files> pairs by pairing the files modified in a commit that closes an issue, to that issue.

Based on the work of Huo et al., we believe that it would be interesting to investigate whether a tool can be developed that uses deep learning to match issues with files on GitHub. We think that such a tool would be beneficial for a large number of projects.

<sup>1</sup><https://github.com/symfony/symfony/>

<sup>2</sup><https://github.com/fchollet/keras>

<sup>3</sup><https://help.github.com/articles/closing-issues-using-keywords/>

GitHub supports the concept of giving a PR or issue one or multiple labels. By default, GitHub includes labels such as *bug*, *enhancement*, *help wanted* and *wontfix*, however, these labels can be customized per project. Large projects often use these labels extensively; Moby<sup>4</sup>, for example uses more than 40 labels to label issues and PRs. Currently, these labels have to be assigned manually which requires a notable amount of work. We believe that it could be using deep learning, or machine learning in general, it should be possible to automatically assign at least some of these labels.

---

<sup>4</sup><https://github.com/moby/moby>

# 7

## Conclusion

We conclude this thesis by answering the research question stated at the beginning of this thesis:

**RQ** What are the benefits and perils of using deep learning in repository mining research?

We answered this question by doing two case studies. In chapter 3 we trained models to solve the merge prediction task and showed that DNNs trained on raw data can perform comparable or even better than traditional machine learning models trained on features conceived by a researcher.

In chapter 4 we collected a dataset of similar PRs using the unsupervised deep learning technique Word2Vec and trained models to find similar PRs using deep learning techniques. We showed that these models are able to generalize to projects they are not trained on. These models, with a precision and recall of respectively almost 70% and approximately 60%, all performed better than the non-neural baselines.

In chapter 5 we explain that the power of neural models comes at a cost. Firstly, Optimizing neural models is difficult because of the high number of hyperparameters and possible architectures. We recommend repository mining researchers to take into account architectures that perform well in other domains, such as NLP, before creating novel architectures. It is also recommended to use random search or Bayesian optimization, instead of grid search when attempting to optimize the hyperparameters. Secondly, Explaining which features a neural model learned to recognize is still extremely difficult, despite techniques being developed. Finally, training neural models is costly, due to the complexity of these models and the large datasets needed. It is therefore important to include a less costly baseline when using neural models in research, to show that the power and thereby the cost of neural models is justified.



# Bibliography

- [1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, *ImageNet Large Scale Visual Recognition Challenge*, *International Journal of Computer Vision (IJCV)* **115**, 211 (2015).
- [2] Y. LeCun, Y. Bengio, and G. Hinton, *Deep learning*, *Nature* **521**, 436 (2015).
- [3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *Imagenet classification with deep convolutional neural networks*, in *Advances in neural information processing systems* (2012) pp. 1097–1105.
- [4] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al., *Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups*, *IEEE Signal Processing Magazine* **29**, 82 (2012).
- [5] S. Wang, T. Liu, and L. Tan, *Automatically learning semantic features for defect prediction*, in *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16 (ACM, New York, NY, USA, 2016) pp. 297–308.
- [6] C. S. Corley, K. Damevski, and N. A. Kraft, *Exploring the use of deep learning for feature location*, in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2015) pp. 556–560.
- [7] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016) pp. 770–778.
- [8] H. Zhang, T. Xu, H. Li, S. Zhang, X. Huang, X. Wang, and D. Metaxas, *Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks*, arXiv preprint arXiv:1612.03242 (2016).
- [9] M. Allamanis, H. Peng, and C. Sutton, *A convolutional attention network for extreme summarization of source code*, in *Proceedings of The 33rd International Conference on Machine Learning* (2016) pp. 2091–2100.
- [10] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, *On the naturalness of software*, in *Software Engineering (ICSE), 2012 34th International Conference on* (IEEE, 2012) pp. 837–847.
- [11] G. Gousios, M. Pinzger, and A. v. Deursen, *An exploratory study of the pull-based software development model*, in *Proceedings of the 36th International Conference on Software Engineering*, ICSE (ACM, New York, NY, USA, 2014) pp. 345–355.
- [12] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, and P. Devanbu, *Cohesive and isolated development with branches*, in *International Conference on Fundamental Approaches to Software Engineering* (Springer, 2012) pp. 316–331.
- [13] G. Gousios, M. Pinzger, and A. v. Deursen, *An exploratory study of the pull-based software development model*, in *Proceedings of the 36th International Conference on Software Engineering* (ACM, 2014) pp. 345–355.
- [14] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, *The promises and perils of mining github*, in *Proceedings of the 11th working conference on mining software repositories* (ACM, 2014) pp. 92–101.
- [15] E. Van Der Veen, G. Gousios, and A. Zaidman, *Automatically prioritizing pull requests*, in *Proceedings of the 12th Working Conference on Mining Software Repositories* (IEEE Press, 2015) pp. 357–361.

- [16] V. Nair and G. E. Hinton, *Rectified linear units improve restricted boltzmann machines*, in *Proceedings of the 27th international conference on machine learning (ICML-10)* (2010) pp. 807–814.
- [17] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, *On the properties of neural machine translation: Encoder-decoder approaches*, arXiv preprint arXiv:1409.1259 (2014).
- [18] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, *Handwritten digit recognition with a back-propagation network*, in *Advances in neural information processing systems* (1990) pp. 396–404.
- [19] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, *Going deeper with convolutions*, in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015) pp. 1–9.
- [20] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, *A convolutional neural network for modelling sentences*, arXiv preprint arXiv:1404.2188 (2014).
- [21] W.-t. Yih, K. Toutanova, J. C. Platt, and C. Meek, *Learning discriminative projections for text similarity measures*, in *Proceedings of the Fifteenth Conference on Computational Natural Language Learning* (Association for Computational Linguistics, 2011) pp. 247–256.
- [22] Y. Kim, *Convolutional neural networks for sentence classification*, arXiv preprint arXiv:1408.5882 (2014).
- [23] W. Yin, K. Kann, M. Yu, and H. Schütze, *Comparative study of cnn and rnn for natural language processing*, arXiv preprint arXiv:1702.01923 (2017).
- [24] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, *Signature verification using a “siamese” time delay neural network*, in *Advances in Neural Information Processing Systems* (1994) pp. 737–744.
- [25] G. Koch, *Siamese neural networks for one-shot image recognition*, Ph.D. thesis, University of Toronto (2015).
- [26] D. E. Rumelhart, G. E. Hinton, R. J. Williams, et al., *Learning representations by back-propagating errors*, *Cognitive modeling* **5**, 1 (1988).
- [27] S. Ruder, *An overview of gradient descent optimization algorithms*, arXiv preprint arXiv:1609.04747 (2016).
- [28] T. Tieleman and G. Hinton, *Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude*, COURSERA: Neural networks for machine learning **4** (2012).
- [29] D. Kingma and J. Ba, *Adam: A method for stochastic optimization*, arXiv preprint arXiv:1412.6980 (2014).
- [30] A. Karpathy and L. Fei-Fei, *Deep visual-semantic alignments for generating image descriptions*, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015) pp. 3128–3137.
- [31] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio, *Show, attend and tell: Neural image caption generation with visual attention*, in *International Conference on Machine Learning* (2015) pp. 2048–2057.
- [32] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, *Dropout: a simple way to prevent neural networks from overfitting*. *Journal of Machine Learning Research* **15**, 1929 (2014).
- [33] D. Tang, F. Wei, N. Yang, M. Zhou, T. Liu, and B. Qin, *Learning sentiment-specific word embedding for twitter sentiment classification*. in *ACL (1)* (2014) pp. 1555–1565.
- [34] C. N. Dos Santos and M. Gatti, *Deep convolutional neural networks for sentiment analysis of short texts*. in *COLING* (2014) pp. 69–78.

- [35] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, *et al.*, *Google's neural machine translation system: Bridging the gap between human and machine translation*, arXiv preprint arXiv:1609.08144 (2016).
- [36] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, *Convolutional sequence to sequence learning*, arXiv preprint arXiv:1705.03122 (2017).
- [37] L. Yu, K. M. Hermann, P. Blunsom, and S. Pulman, *Deep learning for answer sentence selection*, arXiv preprint arXiv:1412.1632 (2014).
- [38] W. Yin, H. Schütze, B. Xiang, and B. Zhou, *Abcnn: Attention-based convolutional neural network for modeling sentence pairs*, arXiv preprint arXiv:1512.05193 (2015).
- [39] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, *Distributed representations of words and phrases and their compositionality*, in *Advances in neural information processing systems* (2013) pp. 3111–3119.
- [40] Z. S. Harris, *Distributional structure*, *Word* **10**, 146 (1954).
- [41] T. D. Nguyen, A. T. Nguyen, and T. N. Nguyen, *Mapping api elements for code migration with vector representations*, in *Proceedings of the 38th International Conference on Software Engineering Companion* (ACM, 2016) pp. 756–758.
- [42] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, *Exploring api embedding for api usages and applications*, in *Proceedings of the 39th International Conference on Software Engineering* (IEEE Press, 2017) pp. 438–449.
- [43] E. Asgari and M. R. Mofrad, *Continuous distributed representation of biological sequences for deep proteomics and genomics*, *PloS one* **10**, e0141287 (2015).
- [44] Q. Le and T. Mikolov, *Distributed representations of sentences and documents*, in *Proceedings of the 31st International Conference on Machine Learning (ICML-14)* (2014) pp. 1188–1196.
- [45] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, *Toward deep learning software repositories*, in *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on* (IEEE, 2015) pp. 334–345.
- [46] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, *Deep learning code fragments for code clone detection*, in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (ACM, 2016) pp. 87–98.
- [47] X. Gu, H. Zhang, D. Zhang, and S. Kim, *Deep api learning*, in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (ACM, 2016) pp. 631–642.
- [48] S. Wang, T. Liu, and L. Tan, *Automatically learning semantic features for defect prediction*, in *Proceedings of the 38th International Conference on Software Engineering* (ACM, 2016) pp. 297–308.
- [49] R. Gupta, S. Pal, A. Kanade, and S. Shevade, *Deepfix: Fixing common c language errors by deep learning*. in *AAAI* (2017) pp. 1345–1351.
- [50] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, *Combining deep learning with information retrieval to localize buggy files for bug reports (n)*, in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on* (IEEE, 2015) pp. 476–481.
- [51] X. Huo, M. Li, and Z.-H. Zhou, *Learning unified features from natural and programming languages for locating buggy source code*. in *IJCAI* (2016) pp. 1606–1612.
- [52] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, *Suggesting accurate method and class names*, in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (ACM, 2015) pp. 38–49.

- [53] S. Jiang, A. Armaly, and C. McMillan, *Automatically generating commit messages from diffs using neural machine translation*, arXiv preprint arXiv:1708.09492 (2017).
- [54] J. Tsay, L. Dabbish, and J. Herbsleb, *Influence of social and technical factors for evaluating contribution in github*, in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014 (ACM, New York, NY, USA, 2014) pp. 356–366.
- [55] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu, *Wait for it: Determinants of pull request evaluation latency on GitHub*, in *12th Working Conference on Mining Software Repositories*, MSR (IEEE, 2015) pp. 367–371.
- [56] G. Gousios and A. Zaidman, *A dataset for pull-based development research*, in *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014 (ACM, New York, NY, USA, 2014) pp. 368–371.
- [57] G. Gousios, *The ghtorrent dataset and tool suite*, in *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13 (IEEE Press, Piscataway, NJ, USA, 2013) pp. 233–236.
- [58] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, 2016) <http://www.deeplearningbook.org>.
- [59] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, *Cross-project defect prediction: A large scale experiment on data vs. domain vs. process*, in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09 (ACM, New York, NY, USA, 2009) pp. 91–100.
- [60] A. Karpathy, J. Johnson, and L. Fei-Fei, *Visualizing and understanding recurrent networks*, arXiv preprint arXiv:1506.02078 (2015).
- [61] G. Gousios, M.-A. Storey, and A. Bacchelli, *Work practices and challenges in pull-based development: The contributor's perspective*, in *Proceedings of the 38th International Conference on Software Engineering* (ACM, 2016) pp. 285–296.
- [62] R. Řehůřek and P. Sojka, *Software Framework for Topic Modelling with Large Corpora*, in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks* (ELRA, Valletta, Malta, 2010) pp. 45–50, <http://is.muni.cz/publication/884893/en>.
- [63] J. Cohen, *A coefficient of agreement for nominal scales*, Educational and psychological measurement **20**, 37 (1960).
- [64] J. R. Landis and G. G. Koch, *The measurement of observer agreement for categorical data*, biometrics , 159 (1977).
- [65] P. Jaccard, *Distribution de la flore alpine dans le bassin des dranses et dans quelques régions voisines*, Bull. Soc. Vaud. Sci. Nat. **37**, 241 (1901).
- [66] J. H. Lau and T. Baldwin, *An empirical evaluation of doc2vec with practical insights into document embedding generation*, arXiv preprint arXiv:1607.05368 (2016).
- [67] Y. Zhang and B. Wallace, *A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification*, arXiv preprint arXiv:1510.03820 (2015).
- [68] R. Hadsell, S. Chopra, and Y. LeCun, *Dimensionality reduction by learning an invariant mapping*, in *Computer vision and pattern recognition, 2006 IEEE computer society conference on*, Vol. 2 (IEEE, 2006) pp. 1735–1742.
- [69] T. Chen and C. Guestrin, *Xgboost: A scalable tree boosting system*, in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining* (ACM, 2016) pp. 785–794.
- [70] V. Sandulescu and M. Chiru, *Predicting the future relevance of research institutions - the winning solution of the KDD cup 2016*, CoRR **abs/1609.02728** (2016).



- [71] B. Vasilescu, D. Posnett, B. Ray, M. G. van den Brand, A. Serebrenik, P. Devanbu, and V. Filkov, *Gender and tenure diversity in github teams*, in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (ACM, 2015) pp. 3789–3798.
- [72] M. T. Ribeiro, S. Singh, and C. Guestrin, *Why should i trust you?: Explaining the predictions of any classifier*, in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (ACM, 2016) pp. 1135–1144.
- [73] W. Fu and T. Menzies, *Easy over hard: A case study on deep learning*, arXiv preprint arXiv:1703.00133 (2017).
- [74] V. Mnih, N. Heess, A. Graves, et al., *Recurrent models of visual attention*, in *Advances in neural information processing systems* (2014) pp. 2204–2212.
- [75] J. Bergstra and Y. Bengio, *Random search for hyper-parameter optimization*, *Journal of Machine Learning Research* **13**, 281 (2012).
- [76] J. Snoek, H. Larochelle, and R. P. Adams, *Practical bayesian optimization of machine learning algorithms*, in *Advances in neural information processing systems* (2012) pp. 2951–2959.
- [77] E. Yudkowsky, *Artificial intelligence as a positive and negative factor in global risk*, *Global catastrophic risks* **1**, 184 (2008).
- [78] M. D. Zeiler and R. Fergus, *Visualizing and understanding convolutional networks*, in *European conference on computer vision* (Springer, 2014) pp. 818–833.
- [79] L. Arras, F. Horn, G. Montavon, K.-R. Müller, and W. Samek, "what is relevant in a text document?": *An interpretable machine learning approach*, *PloS one* **12**, e0181142 (2017).
- [80] J. P. Winkler and A. Vogelsang, "what does my classifier learn?" *a visual approach to understanding natural language text classifiers*, in *International Conference on Applications of Natural Language to Information Systems* (Springer, 2017) pp. 468–479.
- [81] J. Li, X. Chen, E. Hovy, and D. Jurafsky, *Visualizing and understanding neural models in nlp*, arXiv preprint arXiv:1506.01066 (2015).
- [82] A. Kádár, G. Chrupała, and A. Alishahi, *Representation of linguistic form and function in recurrent neural networks*, *Computational Linguistics* (2017).
- [83] A. Radford, R. Jozefowicz, and I. Sutskever, *Learning to generate reviews and discovering sentiment*, arXiv preprint arXiv:1704.01444 (2017).

Cover image designed by Harryarts/Freepik