Automatic Comment Generation using a Neural Translation Model

Tjalling Haije
10346236

Bachelor thesis
Credits: 18 EC

Bachelor Opleiding Kunstmatige Intelligentie

University of Amsterdam
Faculty of Science
Science Park 904
1098 XH Amsterdam

Supervisor
Dr. E. Gavves
H. Heuer, MSc

QUVA Lab
Faculty of Science
University of Amsterdam
Science Park 904
1098 XH Amsterdam

June 24th, 2016

Abstract

Comments help developers quickly understand the functionality of programming code. Unfortunately, creating these comments remains a labour-intensive task despite various help utilities for software development. In this thesis a new approach to automatic comment generation is proposed: the translation of programming code into comments. By posing the problem as a translation task the model can be more flexible in the generated results, and can be ported easily to other programming languages.

A sequence to sequence model has been implemented to generate annotations for a code fragment, after training on a dataset containing code-annotation pairs. First the neural translation model was applied to a Django dataset to generate pseudocode annotations, achieving an accuracy of 41.6%. Subsequently the model was applied to a newly created code-comment dataset, containing 45367 code-comment pairs extracted from 6 open-source software projects. The model had difficulties with this dataset, as the achieved accuracy was relatively low at 10.7%. However, the performance of this translation approach mainly relies on the dataset, and it still has much room for improvement. In addition have a set of important factors been identified which are important for comment quality.

Contents

1 Introduction

Software development involves the creation, testing and documentation of computer code. Various techniques have been developed to facilitate the programmer during the creation and testing of software, however documenting code with comments remains an labour-intensive task [1][2]. Even though documentation is important for a quick understanding of program code, further facilitating in software maintenance, its labour-intensive nature leads to many software projects with inadequate documentation [1][2].

As such, various studies have explored the viability of automatic comment generation; the majority of these focus on manually creating rules which capture the semantic and structural information of the programming code. Using this information a summary or comment is generated for a code fragment [3][4].

Two major issues in the rule-based approach concern the lack of portability and flexibility of the model. Firstly, the manually created rules are dependant on the specific program language used. The creation of new rules is required for automatic comment generation in new program languages, which is a time-consuming process. Secondly, utilizing set rules results in a fixed style of writing in the comments. Thus, the approach of prior research is not optimal. The absence of a widely used implementation for automatic comment generation could also be attributed to these issues.

Provided the limitations associated with using a rule-based approach, an alternative approach could be to pose the problem as a translation task. A machine translation approach would be more flexible and portable, as all rules are learned automatically from the dataset and thus the model is entirely dependant on the dataset given.

However, far too little attention has been paid to the applicability of machine translation techniques for automatic comment generation. As such, the goal of this thesis is to explore the viability of (neural) machine translation as a method for automatic comment generation. The primary challenge towards this goal is the identification of important factors which determine comment quality. The second, practical challenge is the scarcity of available datasets for this specific task, and that of open-source software projects with adequate comments which can be used to generate a dataset. As the task is posed as a translation task, the result will also be evaluated using the BLEU-score[5]. It is a popular evaluation metric for translation tasks which is meant to resemble human evaluation.

As an intermediate step to automatic comment generation, the model will first be applied to the task of automatic pseudocode generation. Pseudocode is a low-level description of programming code, and likely easier to generate as the difference between the programming code and pseudocode is relatively small.

The hypotheses for this thesis is that a neural machine translation model can be used to generate quality comments, trained on a sufficiently large code-comment dataset.

1.1 Outline

This thesis is divided into 8 chapters: first, chapter 2 discusses relevant research to the field of automatic comment generation, followed by a theoretical foundation which gives an overview of relevant theories and techniques. Chapter 4 describes the datasets, methods and evaluation methods used in this thesis. In chapter 5, the experiments and their results will be presented, followed by an analysis and general discussion in chapter 6. Finally chapter 7 concludes the thesis with an comparison of the hypothesis and acquired results, followed by possible future research in chapter 8.

2 Related Work

2.1 Assistance in program comprehension

This study shares the goal of aiding in program comprehension with prior studies. Program comprehension has been a field of study from as early as the 1980s. Early research focused on lending the developer assistance with generating adequate comments during development [6]. For instance Robillard et al [6] use an interactive method in which the developer is asked for clarification of programming code during development. Other studies focus on facilitating program comprehension using the generation of pseudo-code [7] [8]. A recent study by Oda et al [8] involved the use of a statistical machine translation framework to learn the automatic generation of pseudo-code from a Python dataset, a dataset which is also used in the present study.

However, in contrast to aforementioned studies, the focus of the current study lies with facilitating program comprehension through the automatic generation of comments.

2.2 Generating comments

Equal to program comprehension, the task of comment generation has already been explored in prior studies. A number of approaches pose the task as a content selection and text generation problem, where the high-level semantic and structural information is extracted from the programming code and used to generate a natural language summary [3] [4].

For instance Sridhara et al [3] identified a set of interdependent structural and linguistic characteristics for code categories important to comment generation. Text selection is done by selecting code fragments belonging to these important categories, followed by lexicalization of the code fragments according to specific rules for each code structure, resulting in the generation of a comment. Closely related is the study by Wong et al [4] involving comment generation for code-comment pairs extracted from a programming Question and Answer (Q&A) website.
The method implemented in this study is simpler than the aforementioned methods, which rely on manually generated rules dependant on the specific structural characteristics of the programming language used as training data. In addition to its simplicity, the model implemented in the present study makes less assumptions on the training data resulting in a more flexible model.

2.3 Comment generation as a translation task

An alternative approach to automatic comment generation is to pose it as a translation task, which can be abstracted even further to the translation from formal language to natural language. By employing

machine translation techniques programming code can be effectively translated to the corresponding comment. One major advantage of this approach is the wide range of machine translation techniques which becomes available.

Neural Machine Translation (NMT) has seen a growing body of literature in recent years [9][10][11]. A recent study undertaken by Sustkever et al [10] introduced a new sequence to sequence architecture with performance equal to state-of-the-art phrase-based systems on translation tasks. In a follow-up study, Bahdanau et al [11] introduce an attention mechanism for the sequence to sequence architecture which further improves performance. The attention mechanism enables the neural translation model to identify key elements in the source sentence during translation.

Statistical Machine Translation (SMT) is an alternative machine translation method where parallel corpora are analyzed to build a statistical language model. Sentences are translated by sequentially concatenating phrases with the highest probability, the method such as used in a phrase-based model [12] [13].

An early study on the machine translation from formal language to natural language utilizing a SMT model has been done by Burke et al [14], who implemented a grammatical framework to generate a direct translation from Java APIs to natural language. This interpretation contrasts with the current task in the summarization component. Burke et al generate a direct translation of the complete programming code, instead of a summary or comment capturing the most important information.

A different approach was used by Ling et al [15]. The combination of pointer networks [16], code compression, and a modified form of the attention mechanism introduced by Bahdanau et al [11] resulted in a Latent Predictor Network. Using a mix of natural language and structured specifications the latent predictor network generates programming code.

In addition does the latent predictor network address the need for an ability to copy words directly from the source sentence into the output sentence, an ability equally relevant for the task of automatic comment generation.

Two important themes emerge from the studies discussed: posing automatic comment generation as a translation task is a viable approach, and little research has been done identifying important factors which determine the quality of translation between structural different languages, such as translation from programming code to natural language.

3 Theoretical foundation

To understand the methods used and how they are an improvement to existing automatical comment generation methods, an overview of relevant machine learning techniques will be given as a theoretical background. Starting with basic concepts such as neural networks and deep learning, and continuing to more recent and specific techniques such as recurrent neural networks, long short-term memory networks, and the sequence-to-sequence model which will be used in this thesis.

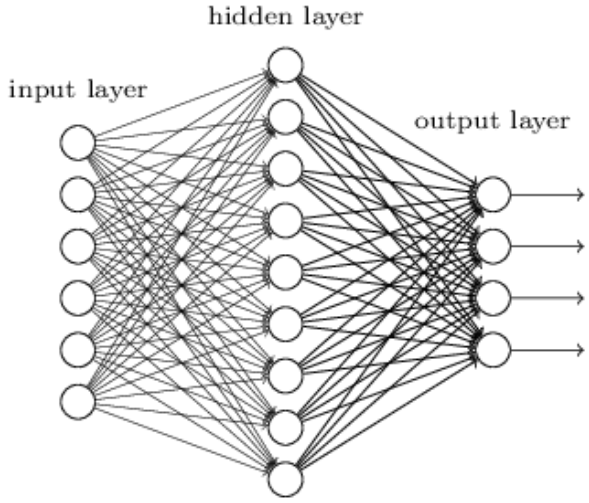

Figure 1: A simple neural network, with 1 input layer, 1 hidden layer and an output layer.[1]

3.1 Neural network

Artificial Neural Networks (ANN), often abbreviated to Neural Networks (NN), are computational models inspired by the biological neural networks seen in animals [17]. Biological neural networks process information through a large quantity of interconnected neurons which process inputs and generate new output values. This type of processing has proven successful for problems which were previously very hard for computers.

The main reason why taking inspiration from the human brain could be beneficial for computers is that there are still tasks easy for humans, which are hard to solve with a computer. Pattern-recognition is a task which humans learn throughout their lives by experience, and the specific knowledge required to recognise a specific person in a crowd of people can be hard to articulate. Thus instead of trying to imbue the product of years of experience as a readymade solution into a computer, it is easier to implement the model for learning and let the computer learn the relevant features itself. Furthermore can computers now be used to solve problems which we don't know how to solve ourselves, and instead let the computer come up with a solution based on large amounts of data.

3.1.1 General structure

A neural network consists of layers of interconnected nodes, as seen in Figure 1. The first layer is used to introduce new data to the model, the final layer is used to output the results. Between the input and output layer there are one or multiple hidden layers, here the actual processing is done. Each node receives input signals from connected nodes, this weighted input signal is converted to the ouput signal using the activation function. For instance when a threshold is used as activation function, the node sends an output of 1 if the signal is above the threshold, or a signal of 0 when below the threshold. This process is similar to neural activation in the human brain.

[1]Retrieved from http://neuralnetworksanddeeplearning.com/chap5.html

Aside from the activation function each node also has a set of weights, which control the connection strength to the other connected nodes. Each time the model generates a wrong answer, the weights are changed according to an learning algorithm.

In addition to the activation function and learning algorithm, are there also architectures which have different interconnection patterns between the layers of nodes. One such variation is the Recurrent neural network, where the network contains cycles. Recurrent neural networks will be discussed in a later section.

3.1.2 Learning strategies

One of the unique abilities of neural networks is the ability to learn from data. Given a certain task the model can learn the optimal function which completes the task. There are three general learning strategies in neural networks:

1. Supervised learning.
   During supervised learning the model learns by examples what the input is and the output should be. The network initially takes a guess to the solution, checks how far it is from the desired result, and adjusts the weights to minimize the error and get closer to the solution. Supervised learning is similar to making homework where a teacher evaluates each answer and gives feedback. Pattern recognition such as handwritten digit recognition falls under this category.

2. Unsupervised learning.
   When there is no dataset with correct answers or the correct answers are not known at all, unsupervised learning can be used. By searching for structure and patterns in the input data a function can be found which is a solution to the task at hand. Since the model does not receive any feedback on the output it generates, it is important to define the task which it needs to model, and any assumptions such as which features are important. This makes the model able to find structure in what is seemingly noise, for instance grouping data in categories such as done in clustering.

3. Reinforcement Learning
   A third form of machine learning is reinforcement learning. Here an agent interacts with its environment, resulting in an observation and some form of punishment or reward according to some unknown factors. When the agent receives a punishment it adjusts the weights to try an other action next time. The goal is to learn to behave in such a way that maximises future rewards. Reinforcement learning is a technique much used in the field of robotics, and can also be used for simulations or games.

3.1.3 Deep learning

Single-hidden-layer neural networks such as used in traditional machine learning work well, but are limited in their capabilities to represent and learn complex tasks. In theory a single-hidden-layer neural network can model any function, given enough hidden units. There could be a hidden unit for every possible input, given the amount of possible inputs is finite. This approach is ofcourse not optimal, as it would result in a very large network with 2^n hidden units, with n the amount of possible inputs.

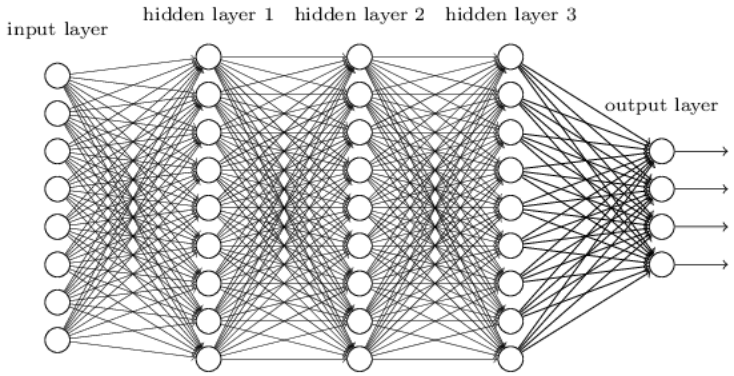

Figure 2: A deep neural network with an input layer, three hidden layers and an output layer.[3]

Deep learning is a class of machine learning algorithms which utilize multiple hidden layers, visualized in Figure 2. The main benefit of multiple hidden-layers is the ability of each layer to learn an increasingly complex set of features. Features of previous hidden layers are combined into more complex features, resulting in a feature hierarchy of increasing abstraction and complexity. This process of automatically extracting features makes deep learning very powerful. An example of the power of deep learning is the recent victory of the Google AI AlphaGo, which used deep learning to find the optimal strategy in playing the game Go[18].

Deep learning was introduced as early as 1965 [19], but has become increasingly popular in recent years. Deep learning requires large amounts of computational power to process all the data and adjust the network accordingly. Large complex networks are required with thousands of hidden units, each with their own set of weights. In recent years the Graphical Processor Units (GPU) have made their entrance to scientific research, giving scientific research and deep learning the boost in computational power it needed[2].

3.2 Recurrent neural network

A Recurrent Neural Network (RNN) is an example of a deep learning architecture. What distinguishes RNNs from other deep learning architectures such as a Convolutional Network[20], is their ability to allow information to persist in the network.

A recurrent network contains loops, also called feedback loops, in which the output for the previous input is used as extra information in processing the next input. It can thus be said that a RNN captures some form of context, or has a form of memory. When a RNN processes t inputs, the network can be visualized as in Figure 3. The specific working of a RNN can be formalized as follows:

$$h_t = f(Wx_t + Uh_{t-1}) \tag{1}$$

[2]http://www.nvidia.com/page/corporate_timeline.html
[3]Retrieved from http://neuralnetworksanddeeplearning.com/chap5.html
[5]Retrieved from http://colah.github.io/posts/2015-08-Understanding-LSTMs/

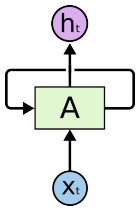

Figure 3: A recurrent neural network. The RNN receives input x_t and the previous output h_{t-1} which are computed to generate a new hidden state h_t.[5]

Here h_t is the hidden state at time step t, W is the weight matrix, U is the transition matrix and x_t the input at time step t. The function f is usually a nonlinear function such as tanh or ReLu. In the example of a translation task would the output vector h_t be a vector with probabilities for all the words in the vocabulary V. The output o_t would then be for instance:

$$o_t = softmax(Vh_t) \tag{2}$$

As a RNN contains loops it can also be 'unrolled', to visualize the processing of all x inputs of the input sequence and the forwarding of the previous output to the next iteration. See Figure 4, here at each step the input is processed using the same network with the previous output as extra information. It can thus be seen as a chain of copies of the same network. However, this analogy is only a virtual one, in reality it is one network which computes all the inputs.

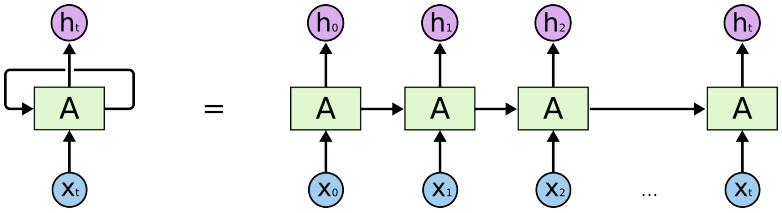

Figure 4: An unrolled recurrent neural network, each cell receives the previous output h_{t-1} as extra input.[7]

The feedback loop in a RNN make it applicable to new tasks. Where standard neural networks require input and output as a fixed-size vector, RNNs can operate over sequences of vectors.

Many tasks depend on sequences in the input, output or both. For instance machine translation can be seen as a sequence of words with interdependent relations, translated to another sequence of words. In Figure 4, the network can be seen as copies of the same network for each word in the sequence. The task of the feedback loop is to add context to the network, by sending information about the previous processed input to the next module, it can memorize earlier outputs when processing the new input and capture dependencies .

[7]Retrieved from http://colah.github.io/posts/2015-08-Understanding-LSTMs/
[8]Retrieved from http://karpathy.github.io/2015/05/21/rnn-effectiveness

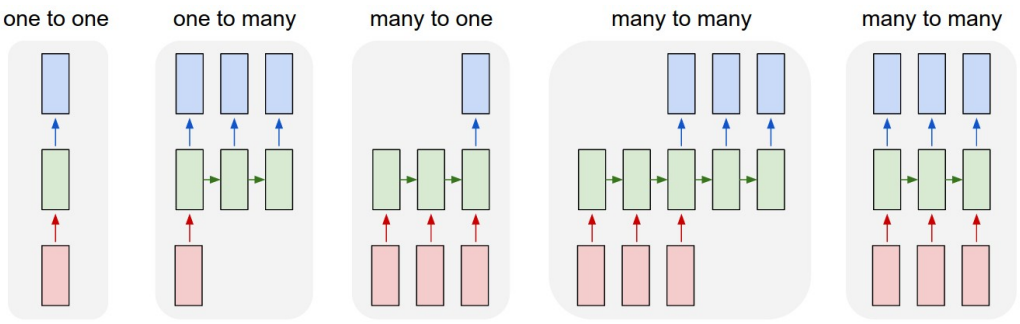

Figure 5: : a standard neural network, an input to output both with fixed dimensions.[8]

A variety of tasks which rely on sequential input, output or both are visualized in Figure 5.

Although in theory RNNs should be able to learn long-term dependencies, in practice the dependencies they can learn are limited. As the gap between the dependent items in the sequence grows, the RNN becomes unable to learn the dependencies. The main cause is the *vanishing gradient problem*[21]. The gradient describes the change in the weights according to the change in error. When the gradient thus becomes extremely small, even a large change has no influence on the network's output, and the model cannot properly learn the change in parameter.

The cause of the vanishing gradient problem lies with the activation functions of the independent cells. For instance, in the first layer of a RNN with multiple layers the input is mapped to a smaller region using the sigmoid activation function. This is repeatedly done for the consequent layers, resulting in a very small number and thus a small change in the final layer. Even a large change gives no significant change in the final layer, as result of the subsequent squashing. Thus when the final item is dependant on the first item, the dependency isn't captured properly because it vanishes after repeatedly being squashed by the activation functions in the cells of several layers[22].

3.2.1 Long short-term memory network

A special type of RNN is the Long Short-Term Memory (LSTM) network, which is able to learn long-term dependencies. Introduced by Hochreiter et al[23] in 1997, the LSTM was proposed as a solution to the vanishing gradient problem. The LSTM contains additional recurrent gates which preserve the error over several iterations, and thus prevents vanishing gradients.

Equal to a RNN does a LSTM network contain feedback loops, and can the network be seen as a set of virtual copies for each item in the input sequence. The main difference lies in the complexity of the recurring module. In a RNN the recurring module calculates the new hidden state by combining the input variable x_t and the output of the previous layer h_{t-1} with a single function, for instance *tanh* or a *sigmoid*. On the contrary a LSTM uses four interacting layers or gates which determine what information to pass on using the feedback loop, and determine the output h_t. In addition uses the LSTM an additional feedback loop called the *cell state*, which contains information from previous

[10]Retrieved from http://colah.github.io/posts/2015-08-Understanding-LSTMs/

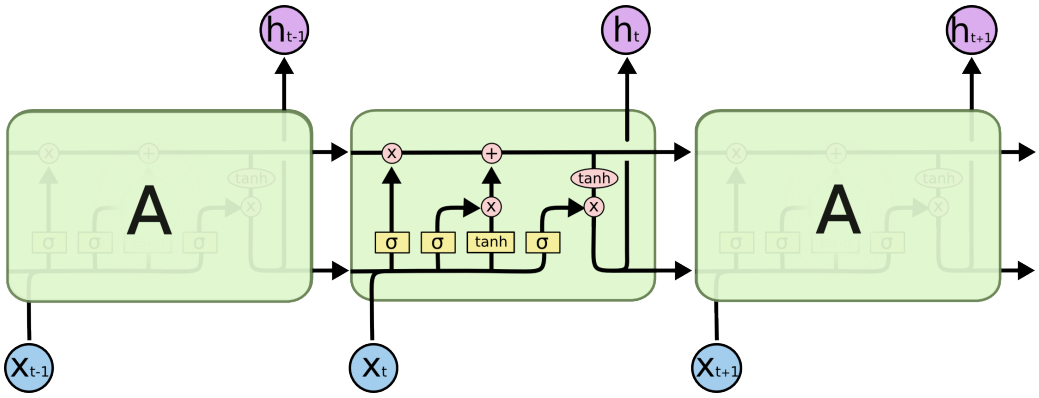

Figure 6: An unrolled Long Short-Term Memory network. The LSTM receives input X_t, the previous output $h_t - 1$, and information from the cell state visualized as the horizontal line at the top in the green cell. The various neural layers or gates, indicated in yellow, determine what information are stored and forgotten from the cell state. The output is generated by combing information from the final *sigma* "output gate" with information from the cell state.[10]

outputs, changed only with minor interactions.

Visualized in Figure 6 is an unfolded LSTM with the additional interacting layers in yellow. Starting from the most left yellow layer, the first σ layer is called the forget gate layer, which determines what information to keep in the cellstate based on the previous output h_{t-1} and the current input x_t. Formalized with W_f the weights of the layer and b_f the biases:

$$f_t = \sigma(W_f\dot{[}h_{t-1}, x_t] + b_f) \tag{3}$$

The next yellow σ layer is called the input gate layer, it generates a matrix i_t which determines what information in the cellstate will be updated. The next *tanh* layer generates a pair of possible values $\tilde{C}$ for the cellstate. These two layers are combined to update the cellstate. Formalized with the individual weights W and biases of b of each layer:

$$\begin{aligned} i_t &= \sigma(W_i\dot{[}h_{t-1}, x_t] + b_i) \\ \tilde{C}_t &= tanh(W_c\dot{[}h_{t-1}, x_t] + b_C) \end{aligned} \tag{4}$$

Now that is determined what will be updated, stored or forgotten from or to the cellstate, the new cellstate is calculated by applying the changes:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{5}$$

Finally, in the last σ layer is determined what information from the previous output h_{t-1} and the current input x_t is used in the output. This information is combined in a *tanh* function with the cellstate C_t to form the new hidden state h_t, the new output:

$$\begin{aligned} o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o) \\ h_t &= o_t * tanh(C)t) \end{aligned} \tag{6}$$

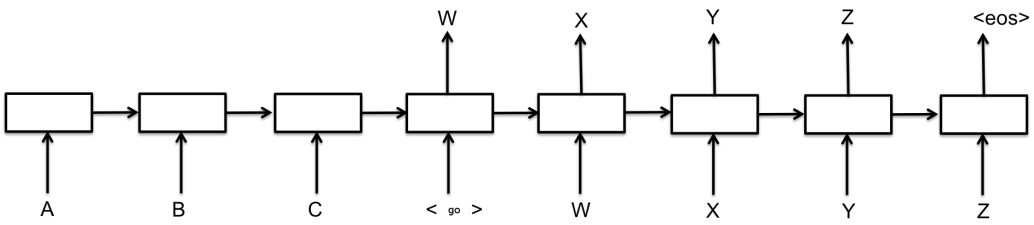

Figure 7: The sequence to sequence model. First the input sequence "A B C" is encoded to a vector, which is then decoded into the output sequence "W X Y Z".[12]

3.2.2 Sequence to sequence model

There are multiple architectures utilizing LSTMs to learn a mapping of sequences to sequences. However most of these methods assume a monotonic alignment between the inputs and outputs, or depend on a known input or output length. Tasks such as machine translation lack these qualifications, where input and output sentences can have varying lengths and alignments. Thus a flexible sequence to sequence architecture is needed.

Sutskever et al [10] introduced a neural architecture which addresses this problem. By utilizing a LSTM to encode the input sentence to a fixed-dimensional vector representation v, and another LSTM to decode the representation v into the output sentence. The second LSTM can be seen as a neural machine translation model such as defined by Mikolov et al [24] [9], trained on the input sentence. This method, visualized in Figure 7, has proven to rival state-of-the-art translation methods [10].

The sequence to sequence model can also be more formally described as; encodation of the the input sequence of vectors $(x_i, ..., x_{t_x})$ by the encoder LSTM into a vector v. The hidden state (output) h_t at every time step t is computed for the input element x_t with the previous hidden state h_{t-1}:

$$h_t = f(x_t, h_{t-1}) \tag{7}$$

The encoded vector representation of the sequence is then defined as the last hidden state of the encoder:

$$v = h_T \tag{8}$$

The decoder is then trained to predict the output sequence $y = (y_1, ..., y_{t'-1})$ word by word. Each word y_t is predicted based on the previous words predicted by the decoder, and the vector v from the encoder. This can be formalized as:

$$p(y) = \prod_{t=1}^{T} p(y_t | y_1, ..., y_{t-1}, v) \tag{9}$$

[12]Retrieved from http://arxiv.org/pdf/1409.3215v3.pdf

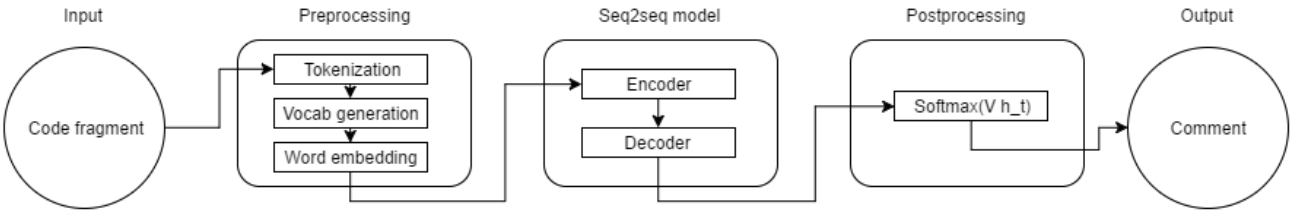

Figure 8: The methodology used in this thesis for automatic comment generation: starting with a code fragment, the data is converted to a sequence of vectors during preprocessing and fed to the neural translation model. After encoding the input sequence to a vector v and decoding the vector into a sequence of vectors representing a comment, the resulting output sequence is converted back to a sequence of tokens during postprocessing, resulting in a comment.

4 Methodology

For the automatic generation of comments a neural translation model has been implemented called a sequence to sequence model [10], already briefly described in Chapter 3: Theoretical Foundation. The model has been applied to two datasets, the first containing code with pseudocode annotations introduced in prior research by Oda et al [8], the second containing code annotated with high-level natural language comments, retrieved from open-source software projects.

The general method used for automatic comment generation can be divided into five steps, visualized in Figure 8. The starting point is a parallel corpus with programming code and corresponding natural language annotations. The acquisition and properties of the datasets used will be discussed in the subsection Datasets. As the programming code and annotations cannot be directly used as input for the machine learning, first the dataset has to be preprocessed. The input sentence is split into smaller parts, called tokens, and each token is converted to a vector notation. This sequence of vectors is much more easy to process by the translation model.

Afterwards the data can be inserted into the sequence to sequence model. This neural machine translation model tries to find an accurate representation for the given input sequence, capturing the structural and semantic information such that it can be accurately translated to a succinct output sequence. The model thus searches for a good representation of a code fragment, such that it can be translated into a natural language annotation such as a comment. The generated output is also a sequence of vectors, during postprocessing these vectors are converted to the corresponding tokens. The final result is a natural language translation of each code fragment.

In the following subsections each step of the methodology from Figure 8 will be further explained.

4.1 Datasets

To get a clear view as to the goal of the study, first the datasets will be presented. This way the input and targeted output becomes clear, and the chosen methods can be more easily explained.

For this thesis two different datasets have been used: the first containing programming code annotated with pseudocode, referred to as the code-pseudocode dataset, the second dataset containing

Property	Code-pseudocode dataset	Code-comment dataset
Total pairs	18805	45367
Test set pairs	1805	4537
Training set pairs	17000	40830
Overlap code train/test set	30.41%	21.36%
Overlap annotations train/test set	32.24%	31.03%
Vocabulary code	6500	47500
Vocabulary annotation	6500	22000

Table 1: Comparison of the code-comment dataset and the code-pseudocode dataset.

programming code annotated with comments, referred to as the code-comment dataset. Pseudo code is a low-level description of the functionality of programming code, and is often more structured than normal code comments making it an easier dataset for the model to train on.

The second dataset is a combination of six open-source software projects from Github[13], each containing well-documented code. The comments in the second dataset give a much more high-level description of the functionality of the programming code, and thus are likely also harder to generate.

Each dataset is divided into a training set containing 90% of the total data, and a test set containing the remaining 10%. Generation of the training and test set is done by shuffling the order of the code-comment pairs, placing 90% in the training set and 10% in the test set. This ensures there are no large discrepancies between the two sets, such as specific type of annotation which occurs a successive number of times locally, thus appearing in only the training or test sets. The properties of the datasets are compared in Table 1.

4.1.1 Code-pseudocode dataset

The first dataset which is used is the Django dataset presented by Oda et al [8]. Django is a well documented Python web framework, and Oda et al have provided each line of source code with a single line pseudocode annotation. All code and pseudocode has been accumulated in two separate files, collectively forming 18805 code-pseudocode pairs. Examples of code-pseudocode pairs are shown in Figure 9.

```
# if self._cull_frequency equals to integer 0.
if self._cull_frequency == 0:

# delete the value under the key key of self._expire_info dictionary.
 del self._expire_info[key]
```

Figure 9: Example code-pseudocode pairs in the Django code-pseudocode dataset.

Of the 18805 code-pseudocode pairs, 17000 random lines were placed in the training set and 1805 in the test set, roughly corresponding to the 90%/10% ratio. The dataset contains multiple similar or equal code-pseudocode pairs: there is a 20.11% overlap of code lines in the training and test set,

[13] https://github.com

and a 19.34% overlap of pseudocode annotations in the training and test set. Examples of simple code-pseudocode pairs occurring multiple times are given in Figure 10.

```
# try,
try :

# if not,
else :
```

Figure 10: Example code-pseudocode pairs in the Django code-pseudocode dataset occuring in the train set as well as the test set.

A major advantage of this dataset is that it is a clean dataset, aside from a few misplaced pseudocode annotations are there hardly any errors which require cleaning. Additionally, the code-pseudocode pairs have a format closely resembling sentence-pairs in a regular natural-language translation task, such as English-to-French translation. Each line of programming code has been annotated with a single line of pseudocode. As pseudocode is a low-level description of the functionality of programming code, the length of sentences are also similar in most cases.

The large difference between pseudocode and normal comments annotations can be said to be the largest disadvantage of this dataset. Pseudocode is a low-level description of programming code, describing every variable, function or action done in the programming code. As such the pseudocode annotation is in general also of roughly the same length as the code line and contains many words directly copied from the code. All these factors are more difficult in code-comment generation. As such, if the model shows good results on this dataset, it is not guaranteed to work equally well on the code-comment dataset. The difference between the datasets is investigated in chapter 5: Experiments and results.

Another issue is the amount of data, which is relatively small for deep learning methods relying solely on the available data. Machine translation techniques rely on large parallel corpora to learn a representation of the input sentence, which can be translated to another language. As an illustration, Sutskever et al [10] trained a sequence to sequence model on the WMT' English to French dataset containing 12 Milion sentences, compared to 18805 pairs in this dataset. A small amount of data can result in overfitting and poor performance on new test data. As the amount of data trained on is relatively small, there are still many types of test sentences and tokens completely unfamiliar to the model. As the amount of training data grows the model learns to represent each of these sentences more accurate, increasing output quality.

4.1.2 Code-comment dataset

As the main focus of this thesis is the automatic generation of comments, a parallel corpus is needed containing code annotated with comments. Unfortunately, at the moment of writing no usable datasets have been found for this task aside from the code-pseudocode dataset. As such a new dataset has been created, consisting of code-comment pairs originating from open source software projects retrieved from Github[14].

[14] https://github.com/

For the creation of this dataset first a top ten of Python software projects were selected on the basis of popularity (number of stars), and project size using the Github advanced search feature. Of each selected project the amount of comments was identified (Block comments starting with #, as well as docstrings starting with """ or ' ' '), and projects containing less than 15.000 comments were rejected. Of the remaining projects the comment quality was inspected by manually inspected a subset of the code-comment pairs. Poor comments are identified as so when uninformative, extremely long or containing redundant information. A few examples are TODO notes, comments containing extensive parameter descriptions, or comments which are in fact commented program code. From this selection the six best projects were chosen:

- Django[15]
- Scikit-learn[16]
- EDX-platform[17]
- Salt[18]
- Pandas[19]
- Pylearn2[20]

The dataset contains a total of 45367 code-comment pairs, after shuffling the order of code-comment pairs 4537 pairs were randomly placed in the test set, and 40830 pairs were randomly placed in the train set, corresponding to the 90%/10% ratio. Other properties of this dataset are shown in Table 1, such as the amount of unique tokens which is much higher than the pseudocode dataset.

4.1.2.1 Dataset generation

Code-comment pairs have been extracted using a python script, which has been made available on Github [21]. The script searches for a unique comment character (# for a block comment, """ or ' ' ' for an docstring) at the beginning of each line. If present it uses the indentation layout of Python code[22] to identify the ending of the comment and the targeted program code. The specific steps for the extraction of block comments with their corresponding code fragments are as described in Figure 11.

As this thesis is an initial exploration of the viability of automatic comment generation using a neural translation model, we focus on small code-comment pairs. The steps described above select only the most specific comments on the lowest indentation level. This measure drops comments describing large pieces of code, such as a description of a complete file or a complete class. The result is a lower

[15]The Django repository on Github is:https://github.com/django/django
[16]The Scikit-learn repository on Github is:https://github.com/scikit-learn/scikit-learn
[17]The EDX-platform repository on Github is:https://github.com/edx/edx-platform
[18]The Salt repository on Github is:https://github.com/saltstack/salt
[19]The Pandas repository on Github is:https://github.com/pydata/pandas
[20]The Pylearn2 repository on Github is:https://github.com/lisa-lab/pylearn2
[21]All code and data used in this thesis is available with documentation on Github: https://github.com/thaije/code-to-comment
[22]Basic knowledge of the Python programming language is assumed. If not the case, reading through the styling guide for Python is recommended: https://www.python.org/dev/peps/pep-0008/
[22]https://github.com/edx/edx-platform/blob/master/openedx/core/operations.py

```
Retrieve a list of file paths to all files containing a block comment.
Loop through each line of each of the files:
    If the line starts with an comment:
        Append each successive line starting with \# to the comment
            If an empty line is encountered immediately after the comment
                Omit the comment
                Continue searching for the next comment

        Append each successive codeline to the code fragment
            If an empty line is encountered after a codeline or the End Of File (EOF)
                Save the comment-code pair
                Continue searching for the next comment

            If code is encountered on a higher indentation level
                Save the comment-code pair
                Continue searching for the next comment

            If a block comment or docstring is encountered on a lower indentation level
                Omit the comment-code pair
                Continue with the newly encountered comment
```

Figure 11: Pseudocode describing the extraction of code-comment pairs from a software project, in this case the script searches for block comments starting with the # character.

variation in code fragment length of code-comment pairs, and on average a lower code fragment length. In Figure 12 an example is visualized.

Extraction of the docstrings with the corresponding code is done in a similar fashion to 11, the only difference being the comment extraction. Docstrings are identified as starting with """ or ' ' ', and ending with the same three tokens. Furthermore are docstrings usually placed after the function or class definition, as such the previous code line is also added to the code if it is an definition. An example can be seen in Figure 12.

After the code-comment pairs have been generated, additional cleaning is done on the dataset. This includes removing code-comment pairs containing "todo" or "to do" (including letter case variations), and removing so called "divider lines" from the comments, which are lines consisting only of repetitions of the "-" or "#" characters.

4.2 Preprocessing

Before the code-comment pairs in each dataset can be used to train the translation model, the input has to be converted to a suitable format for the translation model. Each input and output line is cut into smaller parts (tokenization), and each token is replaced with a number indexing the corresponding token. Furthermore is each word represented as a word embedding (a vector), which is fed into the model.

```
1.  def dump_memory(signum, frame):
2.        """
3.        Dump memory stats for the current process to a temp directory.
4.        Uses the meliae output format.
5.        """
6.
7.        ...
8.
9.        # force garbarge collection
10.       for gen in xrange(3):
11.            gc.collect(gen)
12.            scanner.dump_all_objects(
13.                format_str.format("gc-gen-{}".format(gen))
14.            )
```

Figure 12: An example code fragment from the Github repository edx-platform [24].The three dots indicate code without any comments which has been omitted for this example. During the code-comment pair generation the docstring at line 2-5 is saved with the successive code until a line at the indentation level of line 1 is encountered. However, when the block comment at line 9 is encountered, the docstring is omitted and the block comment is saved instead with its successive code.

. , ” ’ : ;) (! ? (space)

Figure 13: Characters at which the line is split during tokenization.

4.2.1 Tokenization

Tokenization is the process of dividing a sentence into smaller parts, this step is required as the used translation model accepts sequences as input, with each sequence a sentence split into tokens. Tokenization has been done by a simple tokenizer, which splits a sentence at spaces and punctuation marks displayed in Figure 13.

Tokenization at these punctuation marks ensures functions and variables are split as well as parameters inside functions, resulting in more meaningful tokens which can be used during training. The following code fragment serves as an example:

```
textLine.split(character)
```

The codeline above is split six times, at the characters defined in Figure 13. The result is:

```
['textLine', '.', 'split', '(', 'character', ')']
```

In this example ”textLine” and ”split” can be used more easily than ”textLine.split”, to generate a comment such as ”split the textline”.

4.2.2 Vocab generation

An additional step after tokenization is the generation of a vocabulary. For the programming code and the annotations a separate vocabulary is created, containing all unique tokens encountered in

each set. The main advantage of this step is the possible limitation of the vocabulary size. Training complexity as well as decoding complexity increases proportionally to the increase of unique words [25]. Each unique word in the vocabulary has to be encountered a number of times during the training phase such that the model can learn the correct usage. Words used very little thus cannot be used by the model efficiently, as it has not learned a proper representation of the word and the correct usage. Thus, limiting the vocabulary size can contribute to a lower decoding and training complexity, replacing out-of-vocabulary words with a special "_UNK" token.

4.2.3 Word embedding

Now that the vocabulary has been created, each token in the input sequence can be converted to a vector notation which indexes the specific token in the vocabulary. Each token in the input sequence becomes an vector of size 1 by $vocabulary\_size$, which indexes the specific token in the vocabulary.

4.3 Algorithm implementation

The automatic generation of comments has been done using various models and architectures in prior research, such as described in Section 2: Related work. For this thesis the task of comment generation has been posed as a translation task, and a neural translation model has been used to translate programming code into comments.

4.3.1 Sequence to sequence attention model

A sequence to sequence model is a neural machine translation model which utilizes two Long-Short Term Memory (LSTMs) to encode an input sentence to a vector v, and decode the sentence into another language. The basics have already been described in Section 3: Theoretical foundation. Here the specific implementation used will be described.

The general method is identical to the application of a sequence to sequence model to any other translation task. In the case of automatic comment generation: a code fragment is tokenized and each token converted into a vector notation. This sequence of vectors is fed into the encoder and encoded into a vector v. From here the other LSTM decodes the vector into the desired language.

The sequence to sequence model implemented in this thesis differs from the basic implementation in Section 3 in a number of ways, as the decoder has been extended with a attention mechanism, bucketing has been used for efficiency, the input sentences are reversed and beam search has been used in various models. Each point is briefly discussed.

4.3.1.1 Attention mechanism

The most important difference between the basic sequence to sequence model described in section 3 and the implemented model, is the used attention decoder. Introduced by Bahdanau et al, the attention decoder addresses the problem experienced in encoder-decoder models with long sentences[11]. As the input sequence is encoded into a vector of fixed-length, all necessary information has to be compressed into the vector. However, for long sentences this can prove a problem as not all information may fit

into the vector. The attention model addresses this problem by letting the decoder search in the input sequence at every decoding step. To be more precise: at each generated token the decoder searches for the positions with the most relevant information, and uses the context vectors c_t associated with these positions together with all previously predicted words $(y_1, ..., y_{t-1})$ to predict the next word y_t. This can also be written as:

$$p(y_t|y_1, ..., y_{t-1}, x) = g(y_{t-1}, h_t, c_t) \tag{10}$$

Where h_t is the LSTM hidden state for time step t, which is computed by:

$$h_t = f(h_{t-1}, y_{t-1}, c_t) \tag{11}$$

The context vector c_t contains information on the whole input sentence, but strongly emphasizes the information around the h_t position in the sequence. The context vector contains information of tokens at the important positions in the sequence.

The attention mechanism gives the decoder the ability to focus on specific information in the input sentence, which frees the encoder with the burden to encode all information perfectly in the fixed-size vector.

4.3.1.2 Bucketing

In addition to the attention mechanism has a special measure been added to improve efficiency: bucketing. Normally a separate sequence to sequence model has to be created for each input-output pair of a specific length, instead by padding code-comment pairs to the same length with a special "PAD" token only a single model has to be created. However, as this is inefficient for short sentences which would contain many useless "PAD" tokens, a set of buckets are used as a compromise. Code-comment pairs are placed in the smallest bucket in which they fit, and are padded to the size of the bucket. This way only n models have to be created, with n the number of buckets. The buckets used in the model are:

```
(code fragment length, comment length): (5, 10), (10, 15), (20, 25), (40, 50)
```

For example: a code comment pair with a code fragment containing 38 tokens and a comment containing 23 tokens does not fit in the first three buckets, and is thus placed in the fourth bucket and padded to a code fragment length of 40 and comment length of 50.

4.3.1.3 Reversing source sentences

In addition to the encoder-decoder model which was introduced by Sutskever et al [10], they also discovered the significant performance increase as the result of reversing the source sentences. This effect is most likely caused by short-term dependencies which are introduced with the reversal of the source sentence. By reversing the source sentence the first words of the source sentence are much closer to the first words of the output sentence, making their dependencies stronger.

4.3.1.4 Beam search

During the decoding step in the sequence to sequence model, normally a greedy decoder is used to predict the next token in the output sequence. However, this is usually not optimal and it may not lead to the optimal output sequence. Beam search is a method often used to improve the performance of sequence generating models such as the sequence to sequence model[26] [27]. This method is a variation on the Breadth-First search algorithm, a search algorithm which is guaranteed to find the optimal solution or shortest path (in this case the sequence of tokens with the highest total probability). It does so by creating a search tree and searching in a breadth-first fashion, expanding the leaf nodes until a goal state is found. A large disadvantage is that the the memory usage increases exponentially with each layer, and the algorithm runs out of memory before finding a solution on large search tasks.

Beam search is a variation on this technique which optimizes memory usage while striving to achieve the same optimal solution. Beam search limits the amount of nodes at each level in the search tree to a number b, called the beam width. The b nodes are selected at each layer using a heuristic function h which only selects the most promising nodes. The heuristic function allows the algorithm to select nodes leading to a goal state, while the beam width limits the amount of saved nodes and thus limits the memory usage making it able to always reach a solution. The only drawback of beam search is that the optimal solution is not necessarily found, as it depends on the heuristic function to predict promising nodes.

The practical result of this addition is that the model is able to capture additional contextual information. Dependencies between tokens in a sequence are more accurately captured, even negative dependencies. For instance, using beam search unwanted repetition of the same word in a sequence will occur less frequently, as the probability of the same word occurring after each other is relatively low and most probable not the optimal solution.

4.4 Postprocessing

After the encoder has converted the input sequence into a vector and the decoder has successfully decoded the sentence, the result is an sequence of vectors. Each vector contains the probabilities for each token in the vocabulary. Thus finding the output sequence can be formalized as finding the token with the max probability for every output vector y_t in the ouput sequence $(y_1, ..., y_t)$, where finding the output token with the max probability for an output vector y_t can be described as:

$$o_t = softmax(Vy_t) \tag{12}$$

5 Experiments and results

The methodology described under section 3 is applied twice to each dataset, resulting in a model which automatically generates annotations given a code fragment. First is a training phase, the model is trained on the training se containing 90% of the total parallel corpus. During training each generated output value is compared to the desired output, adjusting and improving the model with each iteration. During the second phase the model is tested on the testset containing residual 10%, feeding the model with code-comment pairs which were not present in the training set. Using the BLEU-score

Property	Code-pseudocode dataset	Code-comment dataset
LSTM layers	2	3
LSTM hidden state size	512	512
Vocabulary size code	5000	40000
Vocabulary size annotations	5000 %	20000%

Table 2: Training parameters for each dataset. All parameters have been optimized, except for the LSTM layers and hidden state size for the code-comment dataset, which could not be enlarged due to memory limitations.

and accuracy the generated comments are evaluated, these evaluation methods are discussed in section 5.4: Evaluation Methods.

5.1 Training Details

Although a clean dataset is required for the LSTM sequence to sequence model to produce high-quality results, it is also a very potent model which is fairly easy to train. As the two datasets differ in size, the optimal training parameters also vary slightly for each dataset.

The sequence to sequence model which has been used is an implementation in Tensorflow, based on an existing implementation[25]. Testing was executed on a Nvidia Titan X GPU, which was shared with three other students. A Troch7 implementation[26] has also been used with identical settings to identify the possible increase in comment quality when beam search is applied.

The training parameters with equal values for both datasets include the parameter initialization, training iterations, learning rate, batch size, model implementation and bucket size:

- All parameters of the LSTMs were initialized with the uniform distribution between -0.1 and 0.1.
- Each model was trained for a total of 30 epochs.
- Gradient descent was used with a learning rate of 0.5. The decay factor was set to 0.99, such that 300 training iterations without improvement results in a decrease in learning rate of 1%.
- Batches of size 64 were used during training.
- The model has been tested with a beam width of 7 or no beam search.
- The buckets in which sentences of different length were placed are:

  ```
  (code fragment length, comment length) (5, 10), (10, 15), (20, 25), (40, 50)
  ```

  Due to the exponential increase in memory usage as the result of enlarging the buckets or adding an extra bucket, the used GPU could not handle larger or more buckets.

[25]Sequence to sequence model in Tensorflow: https://www.tensorflow.org/versions/r0.8/tutorials/seq2seq/index.html
[26]A sequence to sequence model implementation in Torch7: https://github.com/harvardnlp/seq2seq-attn

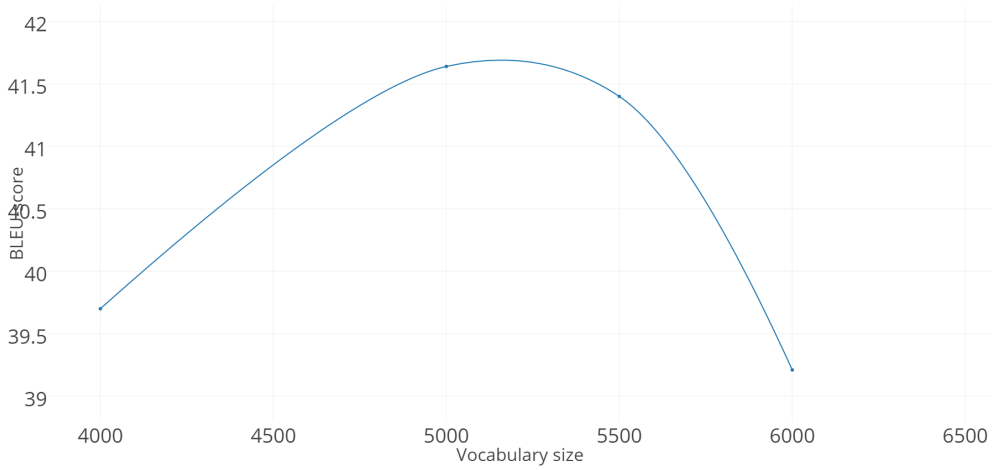

Figure 14: Optimalization of the vocabulary size of the sequence to sequence model. A model with 2 LSTM layers each, hidden state size of 512 and no beam search was applied to the Code-Pseudocode dataset with different vocabulary sizes to find the optimum value.

Aside from these settings have the vocabulary size, LSTM hidden state size and amount of LSTM layers also been optimized, visualized in Table 2. The maximum size of the model which has been tested with is 3 LSTM layers and a hidden state size of 512, further enlarging was not possible due to memory limitations. In Figure 14 the optimalization of the vocabulary size has been illustrated.

5.2 Baseline

Results from a phrase-based model [28] and a sequence to sequence model with an alternative attention mechanism introduced by Ling et al [15] are used as a baseline for the code-pseudocode dataset. These results originate from the study done by Ling et al [15].

As the code-comment dataset has been newly introduced in this study, no baselines are available for this dataset. In addition was the implementation of another baseline method for this dataset unsuccessful.

5.3 Evaluation

The metrics that are used to evaluate the models are the BLEU-score, accuracy and perplexity. Perplexity is used during training to monitor the average negative log probability, which is minimized as the training objective. The BLEU-score and accuracy are used to evaluate generated output, such as comments or pseudocode, during the test phase.

5.3.1 BLEU-score

The main evaluation method used in this study is the BLEU-score [5]. Specifically designed as an evaluation metric for machine translations, the BLEU-score is an inexpensive, fast, language-independent

method which correlates highly with human evaluation. The essential idea is that the best machine translation possible is one closely related to the translation of an human expert. As such the BLEU-score analyzes the closeness to a reference translation.

Factors which determine the BLEU-score are the co-occurence of n-grams in the generated and reference translation, and a brevity penalty which penalizes a difference in sentence length.

The modified n-gram precision used in the BLEU-score can be formalized as:

$$p_n = \frac{\sum_{C \epsilon \{Candidates\}} \sum_{n-gram \epsilon C} Count_{clip}(n-gram)}{\sum_{C' \epsilon \{References\}} \sum_{n-gram' \epsilon C'} Count_{clip}(n-gram')} \tag{13}$$

The BLEU-score can be calculated on the basis of multiple references translations. However, the datasets used in this thesis contain only a single correct reference, as such the formula can be simplified to:

$$p_n = \frac{\sum_{n-gram \epsilon Candidate} Count_{clip}(n-gram)}{\sum_{n-gram' \epsilon Reference} Count_{clip}(n-gram')} \tag{14}$$

This corresponds to the co-occurrence of n-grams in the generated translation and the reference. The n-grams used are 1-grams, 2-grams, 3-grams and 4-grams. An example as to the co-occurrence of these n-grams:

```
Generated comment: Delete item from the list
Reference comment: Delete the item from the list
```

The n-gram precision of the generated comment compared to the reference comment would then be:

- 1-gram precision: 5/6 (”the” missing)
- 2-gram precision: 3/5 (”Delete the”, ”the item” missing)
- 3-gram precision: 2/4 (”Delete the item”, the item from” missing)
- 4-gram precision: 1/2 (”Delete the item from” missing)

The BLEU-score thus indirectly penalizes generated sentences which are too short, as the n-gram precision is lower. On the other hand a disadvantage also becomes clear, the missing of a word such as ”the” is penalized even though the word is not critical for the comment. In this example two individual sentences are stated, however in practice the n-gram precision is calculated over the complete test file.

The BLEU-score also contains a brevity penalty to penalize sentences which are too long:

$$BP = \begin{cases} 1 & if c > r \\ e^{(1-\frac{r}{c})} & if c \leq r \end{cases}$$

Here r is the reference sentence length, and c the length of the candidate sentence.

Combining the n-gram precision with the brevity penalty becomes:

$$BLEU = BP * \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{15}$$

With p_n the n-gram precision of n-grams up to N, and positive weights w_n summing up to one. In this thesis the BLEU-score has been calculated with $N = 4$ and uniform weights $w_n = \frac{1}{N}$. Furthermore has the BLEU-score been calculated over the complete test file.

As the sequence to sequence model generates a sequence of token, each token is divided by a space. To prevent the BLEU-score from excessively penalizing the model for empty spaces, a spaced test file was used to calculate the score. The test file was tokenized as described in section 4.3, and between each token a space was inserted if none present yet.

5.3.2 Accuracy

In addition to the BLEU-score will generated comments be evaluated using the accuracy. Accuracy is formalized as follows:

$$Accuracy = \frac{true\_positives}{true\_positives + false\_positives} \tag{16}$$

In the case of a translation task we are interested in how many of the generated comments are exactly correct, the accuracy rephrased becomes:

$$Accuracy = \frac{correct\_generated\_comment}{correct\_generated\_comment + incorrect\_generated\_comment} * 100 \tag{17}$$

The result is multiplied with 100 to convert the result into a percentage between 0 and 100. As the accuracy can only identify the percentage of entirely correct generated comments it is less informative than the BLEU-score, as it does not give an indication of generated comments which are close to correct.

The main use for the accuracy is the identification of problems in the implementation of the BLEU-score or the translation model; an accuracy of zero likely indicates an error in the translation model. Another use of accuracy is the identification of unique correctly generated comments; of each dataset the overlap between the training and test set been identified, if the accuracy is higher than the overlap this indicates the model has learned the training examples to such a degree that they can be combined to form new correct comments.

5.3.3 Perplexity

The final metric which has been used is the perplexity. Perplexity is a metric which indicates how well a model predicts a sample. In contrast to the accuracy and BLEU-score the perplexity has been solely used to identify the progression of the model during training. The sequence to sequence model minimizes the average negative log probability (p_{target_i}) of the target words ($target_i$) as training goal, formalized:

$$loss = -\frac{1}{N}\sum_{i=1}^{N} \ln p_{target_i} \tag{18}$$

The typical measure used for machine translation tasks is the average per-word perplexity, which can be formalized as:

$$Perplexity = e^{-\frac{1}{N}\sum_{i=1}^{N} \ln p_{target_i}} = e^{loss} \tag{19}$$

5.4 Results

	Code-Pseudocode		Code-Comment	
	BLEU	Accuracy	BLEU	Accuracy
Phrase-based	47.6	31.5	-	-
Adapted seq2seq	58.9	38.8	-	-
Seq2seq attention 3x512	40.4	27.1	7.5	10.7
Seq2seq attention 2x512	41.6	26.8	5.1	8.3
Seq2seq attention 2x512 beam 7	52.6	41.4	-	-

Table 3: Comparison of the code-comment dataset and the code-pseudocode dataset. 2x512 and 3x512 indicate the amount of LSTMs layers, followed by the LSTM hidden state size.

The results are reported in Table 3. In the table Adapted seq2seq is the sequence to sequence model as described by [15]. Seq2seq attention is the model implemented in this thesis, with seq2seq beam 7 the sequence to sequence model with beam search and a beam of 7. On the Code-Pseudocode dataset the phrase-based model and the adapted sequence to sequence model outperform the standard sequence to sequence model implemented in this thesis. The phrase-based model generates good results as it generates only annotations equal to the length of the correct annotation, giving a large boost to the BLEU-score. The adapted sequence to sequence model works well because of the additional attention mechanism introduced by Ling et al [15] which gives the model additional contextual information during translation.

With the addition of beam search the model performs comparable or bettern than the baselines, the extra contextual information from the beam search resulting in a significant performance improvement. Furthermore shows the seq2seq greater results a simpler configuration consisting of 2 layers in each LSTM, compared to 3 layers in each LSTM. This can be contributed to the small size of the code-pseudocode dataset, which contains not enough data to fully train a more complex configuration.

On the other hand performs the sequence to sequence model much worse on the Code-Comment dataset, where a higher level of comprehension of the programming code is required for the generation of a qualitative comment. The best result is achieved by a slightly more complex model with 3 layers in each LSTM. In Table 4 the Unique Correctly Generated Annotations (UCGA) are listed for the best performing model configuration of each dataset. Here the difference becomes even more apparent: on the code-pseudocode dataset 32.67% of the correctly generated annotations are unique; meaning these do not occur in the training file and have been the result of the learning process of the model.

Dataset	Model	Overlap	UCGA
Code-Pseudocode	Seq2seq attention 2x512 beam 7	32.2	32.76
Code-Comment	Seq2seq attention 3x512	31.0	0.8

Table 4: Comparison of the generated annotations in the code-comment dataset and the code-pseudocode dataset. The UCGA is an abbreviation for Unique Correctly Generated Annotations. These are annotations which are not directly copied from the training set (not in the training / test-set overlap, and thus are generated on the basis of training items.

```
# call the method self . connection . ehlo .
self . connection . ehlo ( )
BLEU = 100.00

# call the method msgs . decode with an argument <unk> , substitute the result for msgs .
msgs = msgs . decode ( 'utf-8' )
BLEU = 69.43

# call the method MIMEText . __init__ with 5 arguments : self , text , subtype and None .
MIMEText . __init__ ( self , text , subtype , None )
BLEU = 71.60
```

Figure 15: Three examples of generated pseudocode for a line of code from the Code-Pseudocode dataset, along with the corresponding BLEU-score. Incorrect segments are marked in red. Generated with the Seq2seq attention 2x512 beam 7 model.

In comparison: only 0.8% of the correctly generated annotations for the code-comment dataset are newly generated by the model..

5.4.1 Evaluation

Examples of the annotations generated for programming code of the Code-Pseudocode and Code-Comment dataset are illustrated respectively in Figure 15 and Figure 16. From the results in Figure 15 it is clear that the model can successfully copy tokens directly from the programming code into the annotation. The majority of the errors originate from inaccuracies in counting objects, copying tokens directly from the programming code into the annotation, and out-of-vocabulary words. Although new annotations can be generated by the model as displayed in Table 4, these are only correct for combinations of annotations encountered in the training set. New annotations radically different from seen ones tend to be generated incorrectly.

The generated comments for fragments of programming code in the Code-Comment dataset are illustrated in Figure 16. A small set of code-comment pairs occurred multiple items in the training and test set, which were also correctly translated (see first example). However the majority of the generated comments contained only a small set of relevant keywords, or were completely unrelated. For instance the second example: the keyword generate is correctly filtered from the code, however in combination with the prior tokens a incorrect comment is formed: stating the opposite of the correct comment. Other generated comments such as example 3, were completely unrelated to the code fragment.

The large discrepancy in performance between these two datasets can be derived from the difference between the datasets themselves. The Code-Pseudocode dataset is a very consistent dataset in which

```
# Default to ZeroMQ for now
ttype = 'zeromq'
BLEU = 100

# Should be able to generate (Original comment: Can no longer regenerate certificates for the user)
response = self._generate( course_key=self.EXISTED_COURSE_KEY_2,
username=self.STUDENT_USERNAME )
self.assertEqual(response.status_code, 400)
BLEU = 0.00

Original: Set a minimum threshold of 0.25
# Check that we can be to least the warning on two iteration
sfm = SelectFromModel(clf, threshold=0.25)
sfm.fit(X, y)
n_features = sfm.transform(X).shape[1]
```

Figure 16: Three examples of generated comments for code fragments from the Code-Comment dataset, along with the corresponding BLEU-score. Incorrect segments are marked in red. Generated with the Seq2seq attention 3x512 model.

all the annotations have been manually created: the same codeline always has the same annotation. As a result the amount of unique tokens is also very low.

This cannot be said for the Code-Comment dataset; the dataset is the result of combining 6 different software projects which each have a slightly different style in their comments. The wording consistency varies heavily as a result, and the amount of unique tokens is almost tenfold. To filter all important information and accurately convert the programming code into a high-level description, a lot of data is needed for the model to learn which factors and features are important. However, the model is not able to do so, thus the complexity of this dataset is too large in comparison to its size. Another difference is the need for extra contextual information in some comments. Many comments do not only refer to the directly annotated programming code, but also to code somewhere else in the file. In the current implementation this is not possible with the sequence to sequence model.

6 Discussion

In the conducted experiments, the implemented sequence to sequence model has shown to produce reasonable results on the task of automatic pseudocode generation, scoring a BLEU-score of 41.6 with the default model. The addition of beam search resulted in a significant improvement, generating a BLEU-score of 52.6 rivalling state-of-the-art methods such as a phrase-based model and the sequence to sequence model implemented by Ling et al [15]. In addition to contextual information which improved performance (in the form of beam search), also the vocabulary size has shown to be an important factor, where a balance between complexity and simplicity as the result of variations in vocabulary size gave the best results.

Application of the model on the newly created code-comment dataset resulted in a low score of 7.5. These low results indicate the importance of a number of factors which differentiate the two datasets: Wording consistency, contextual information to code outside of the current code-comment pair, and the balance between dataset complexity and size. The wording consistency is related again to the

complexity, as large variations in comment formalizations results in more unique tokens and a larger dataset complexity.

The original aims of this thesis were to generate quality comments from programming code by approaching it as a translation task, and also to identify important factors which determine comment quality. As an intermediate step the generation of pseudocode from programming code was done, and this proved to be possible. Using the code-comment dataset and the implemented translation model automatic generation of quality comments could not be achieved, however the main causes have been identified. The important factors which influence the comment quality have been identified, such as listed in the previous paragraphs.

The importance of contextual information for comment generation is backed up by prior studies, such as McBurney et al[29] which generate comments for a method solely from the context surrounding it with good results, instead of the code inside the method.

Prior research also acknowledged the problem of generating a large, consistent and clean code-comment dataset[3] [4]. However, the majority of these prior studies focus on the manual creation of rules to extract important structural and semantic information, used to form a comment. The effect of the dataset on the model performance is much less for these methods, when compared to a model relying solely on the dataset. As the model implemented in this thesis is of the latter type, the approach taken in this thesis has the disadvantage when compared to related research.

The main point of discussion is the evaluation of the code-comment dataset. It would be interesting to see what the performance would be of the baseline methods on the code-comment dataset compared to the implemented model, and also the effect of beam search on the performance on the code-comment dataset. Furthermore is the evaluation method a possible point of discussion. The BLEU-score compares the generated comment to a single correct comment. Although a model achieving a perfect BLEU-score would be guaranteed to generate quality comments, in reality there are many possible correct comments for a single code fragment.

Despite these uncertainties, the identified factors important for comment quality are significant for the field of automatic comment generation. In addition does this thesis identify the issues encountered when translating programming code to comments, and various possible improvements to improve performance.

7 Conclusion

The viability of a machine translation approach to automatic comment generation is investigated in this thesis. A neural machine translation model called a sequence to sequence model has been applied to two datasets.

The model achieved a BLEU-score 52.6 on the code-pseudocode dataset introduced by Oda et al[8]. Secondly the model was applied to a newly generated code-comment dataset which resulted in a BLEU-score of 7.4.

In addition have a number of important factors been identified which determine comment quality. First of all is the wording consistency important, where a consistent commenting style is more favorable. The second factor is context: the neighbouring tokens of a token during translation, but also the relations of programming code to other functions and programming code. Thirdly, also the size of the dataset important. The required minimal size of the dataset rises with the increase in dataset complexity. A major factor which determines dataset complexity is wording consistency, were low wording consistency leads to a high number of unique tokens, which increases dataset complexity.

The first conclusion which can be made is thus that pseudocode can be automatically generated for programming code with reasonable quality using the a machine translation approach. The second conclusion is that automatic comment generation using such an approach requires a large consistent dataset to generate quality comments. Thus, as long as generation of a large consistent dataset remains difficult, the current method may not be the optimal approach for this task. The final conclusion is that a successful model for automatic comment generation needs to capture additional contextual information compared to the implemented model in this thesis.

8 Future Work

The sequence to sequence model used in the present study has shown reasonable high performance on automatic pseudocode generation for programming code. A major improvement on the performance on this task could to extend the model with an copy mechanism, allowing it to directly copy a piece of programming code to the pseudocode annotation.

For the automatic generation of comments for programming code the main focus should lie on the generation of a large consistent code-comment dataset, as any neural model or statistical model relies heavily on the dataset. The importance of contextual information also became apparent in the present study, as such future research into providing a model with additional contextual information would be promising. Using the extra information the code fragment can be understood in a context of surrounding programming code and related programming code. Finally, the addition of structural information could also lead to an performance increase. There exist lexical analyzer for programming code which categorize tokens into lexical categories. The information could be combined with word embedding of the specific token, as to provide the model with additional semantic information on each token.

References

[1] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pp. 68–75, ACM, 2005.

[2] M. Kajko-Mattsson, "A survey of documentation practice within corrective maintenance," *Empirical Software Engineering*, vol. 10, no. 1, pp. 31–55, 2005.

[3] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 43–52, ACM, 2010.

[4] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pp. 562–567, IEEE, 2013.

[5] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting on association for computational linguistics*, Association for Computational Linguistics, 2002.

[6] T. E. Erickson, "An automated fortran documenter," in *Proceedings of the 1st annual international conference on Systems documentation*, pp. 40–45, ACM, 1982.

[7] P. N. Robillard, "Schematic pseudocode for program constructs and its computer automation by schemacode," *Communications of the ACM*, vol. 29, no. 11, pp. 1072–1089, 1986.

[8] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pp. 574–584, IEEE, 2015.

[9] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, "Recurrent neural network based language model.," in *INTERSPEECH*, vol. 2, p. 3, 2010.

[10] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, pp. 3104–3112, 2014.

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

[12] D. Chiang, "Hierarchical phrase-based translation," *computational linguistics*, vol. 33, no. 2, pp. 201–228, 2007.

[13] P. Koehn, M. Federico, W. Shen, N. Bertoldi, O. Bojar, C. Callison-Burch, B. Cowan, C. Dyer, H. Hoang, R. Zens, *et al.*, "Open source toolkit for statistical machine translation: Factored translation models and confusion network decoding," in *Final Report of the 2006 JHU Summer Workshop*, 2006.

[14] D. A. Burke and K. Johannisson, "Translating formal software specifications to natural language," in *Logical aspects of computational linguistics*, pp. 51–66, Springer, 2005.

[15] W. Ling, E. Grefenstette, K. M. Hermann, T. Kocisky, A. Senior, F. Wang, and P. Blunsom, "Latent predictor networks for code generation," *arXiv preprint arXiv:1603.06744*, 2016.

[16] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Advances in Neural Information Processing Systems*, pp. 2674–2682, 2015.

[17] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[18] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[19] A. Ivakhnenko, "Polynomial theory of complex systems," *IEEE Transactions on Systems, Man, and Cybernetics*, no. 4, pp. 364–378, 1971.

[20] P. Y. Simard, D. Steinkraus, and J. C. Platt, "Best practices for convolutional neural networks applied to visual document analysis.," in *ICDAR*, vol. 3, pp. 958–962, 2003.

[21] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München," 1991.

[22] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[23] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[24] M. Sundermeyer, R. Schlüter, and H. Ney, "Lstm neural networks for language modeling.," in *INTERSPEECH*, pp. 194–197, 2012.

[25] S. J. K. Cho, R. Memisevic, and Y. Bengio, "On using very large target vocabulary for neural machine translation," 2015.

[26] D. Furcy and S. Koenig, "Limited discrepancy beam search," in *IJCAI*, pp. 125–131, 2005.

[27] A. Graves, "Sequence transduction with recurrent neural networks," *arXiv preprint arXiv:1211.3711*, 2012.

[28] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, *et al.*, "Moses: Open source toolkit for statistical machine translation," in *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*, pp. 177–180, Association for Computational Linguistics, 2007.

[29] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 279–290, ACM, 2014.