Suggesting Descriptive Method Names: An Exploratory Study of Two Machine Learning Approaches

Oleksandr Zaitsev[1,2], Stephane Ducasse[1],
Alexandre Bergel[3], and Mathieu Eveillard[2]

[1] Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL
{oleksandr.zaitsev,stephane.ducasse}@inria.fr
[2] Arolla
mathieu.eveillard@arolla.fr
[3] ISCLab, Department of Computer Science (DCC), University of Chile
abergel@dcc.uchile.cl

Abstract. Programming is a form of communication between the person who is writing code and the one reading it. Nevertheless, very often developers neglect readability, and even well-written code becomes less understandable as software evolves. Together with the growing complexity of software systems, this creates an increasing need for automated tools for improving the readability of source code. In this work, we focus on method names and study how a descriptive name can be automatically generated from a method's body. We experiment with two approaches from the field of text summarization: One based on TF-IDF and the other on deep recurrent neural network. We collect a dataset of methods from 50 real world projects. We evaluate our approaches by comparing the generated names to the actual ones and report the result using Precision and Recall metrics. For TF-IDF, we get results as good as 28% precision and 45% recall; and for deep neural network, 46% precision and 32% recall.

Keywords: Software Evolution · Machine Learning · Method Names.

1 Introduction

The approach to programming has significantly changed in the past few decades. Instructions written by programmers are not solely meant for a computer to execute. Source code must be read by humans in many critical situations (e.g., bug fixing, maintenance) [15]. Developers spend most of their time reading the source code. According to Kent Beck and Robert Martin [6, 18], the ratio of time spent reading versus writing is well over 10 to 1. Making source code easier to read decreases the cost of software development and maintenance. In practice, the readability of source code is often overlooked. Despite understanding the importance of clean code, developers choose poor names for their entities, create long functions and God classes (a well known code smell), fail to write documentation comments [16, 9]. But even good development ethics can not ensure that the system remains clean and comprehensive over time. Software evolves [8], code gets refactored and modified, which often changes the purpose of variables, functions, and classes, as well as the relations between them. Good identifier names will degrade over time, bad ones will become even more misleading. Improving and updating identifier names is one of the key aspects of maintaining an evolving software system. This is a complicated task that requires a profound understanding of the entire system at all times. In the context of large software systems that continue growing in size and complexity [17], such understanding is virtually impossible without the assistance of automated tools. We need tools that will support developers in maintaining the consistency and understandability of the codebase. In this paper, we focus on the quality of method names and study how a descriptive name can be generated from a method's body. We approach this problem as a problem of text summarization and explore two approaches: one based on TF-IDF [19] and the other one using a deep recurrent neural network [23]. These methods were chosen because the first one is a most widely used *extractive approach*, meaning that it generates the summary by extracting the words from the text it is given; and the second one is a state of the art *abstractive approach*, meaning that it can produce words that were not present in the original text. In our case, the summary to generate is the method name, and the original text is the method body. We collected a dataset of 132,046 methods from 50 of real world projects. We cleaned and tokenized this code and used it to train the two models to suggest descriptive names for methods. We evaluate the approaches by comparing method names proposed by our models to the actual names of these methods. Those actual names were given by the programmers so they can be considered the ground truth. After training the models on 70% of methods and evaluating their suggestions on other 20% of methods, we achieved 28% precision, 45% recall with TF-IDF model and 46% precision, 32% recall with a deep learning model. The contribution of this paper are: (i) we argue that programming languages syntax and conventions influence the preprocessing of source code and names, and we propose a methodology for the Pharo programming language; and (ii) we give and compare first results for two text summarization approaches, one abstractive and one extractive.

The rest of this paper is structured as follow: In Section 2 we present the two machine learning models that we considered. In Section 3, we describe the collection, tokenization and filtering of source code and method names. Then, in Section 4 we present the experiment setup used to test the two approaches. We present and discuss the results in Section 5. We close the paper with discussion of related work, Section 6 and the conclusion 7.

2 Two Approaches for Text Summarization

Source code written by programmers has statistical properties similar to the natural languages such as English or Chinese. In fact, code is even more repetitive and predictable than natural languages [12]. Which means natural language processing approaches should be applicable to it. In this context, method names can be seen as summaries of their method bodies in the same way as title is a summary of an article. Therefore, the problem of generating method names can be seen as a problem of summarizing the source code in method's body with a couple of English words. We will just extract all identifiers in a method body, then we split them into words either by CamelCase or on underscores. That way, a method body becomes a document that we want to summarize. This preparation step is described in Section 3. Text summarization is divided into extractive approaches and abstractive ones. We selected one approach from each for our experiment: extractive is done with TF-IDF combined with an n-gram language model; abstractive is done with an attention based sequence to sequence neural network.

2.1 Extractive Model: TF-IDF with N-grams

This approach is based on two steps: first extract the important words with TF-IDF, and then order them with n-gram language model. TF-IDF stands for *Term Frequency-Inverse Document Frequency*, a measure of word importance that works by determining the relative frequency of a word in a specific document compared to the inverse proportion of that word over the entire corpus of documents [19]. Intuitively, this means that the high TF-IDF scores will be assigned to words that are frequent in a given document and rare in all other ones. It allows one to find the most representative words (keywords) in a document and therefore can be used to summarize the given document. TF-IDF score is low for words that frequently occur in the language — such as "and", "the", etc. for English — and high for words that are frequent only in a given document. The word scores are also used to decide how many words there will be in the summary, the naive solution being to fix a threshold score above which words will make up the summary. However, to get a human readable summary of a document, the keywords also need to be ordered in a "natural" way. For this we use the *n-gram language model*, a probabilistic model that learns the probability distribution of n-grams (sequences of n words) which can be used to compute the joint probability of any given sequence of words. For summarization, a simple solution consists in generating all possible permutations of the keywords extracted by TF-IDF, and then find the most likely one. TF-IDF, as used in Information Retrieval, is trained and applied on the same data. To use it on previously unseen data (summarizing new documents) it must be slightly adapted. IDF is computed on the initial dataset, that can be called training dataset. This allows to compute a measure of "surpriseness" of each word within the domain of the training dataset. Then TF is computed on a, possibly new, document to be summarized. This can raise a problem if the new document contains words never appearing in the training dataset. In this case, IDF would be a division by 0. A simple solution is to remove such new words. To avoid a similar problem with n-gram model, words that do not appear both in documents and summaries are removed. If they appeared in the documents but not the summaries, they could be extracted by TF-IDF, but not ordered by the n-gram model.

2.2 Abstractive Model: Sequence to Sequence Neural Network

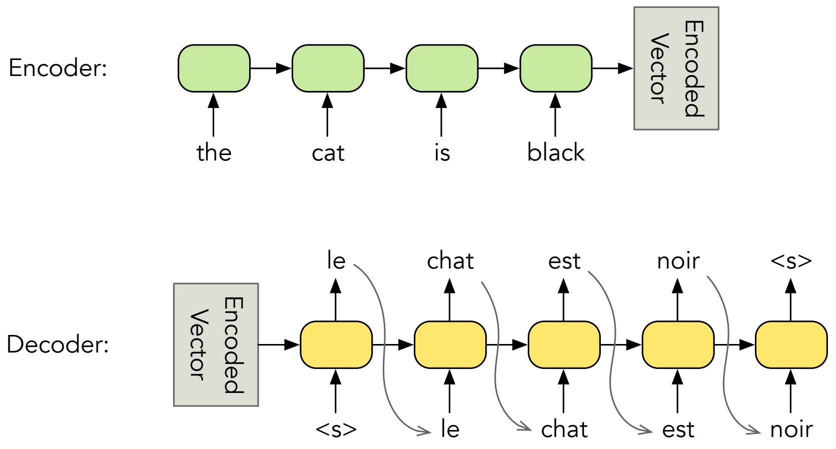

Fig. 1. Using sequence to sequence recurrent neural network to translate an English sentence *"the cat is black"* to a French sentence *"le chat est noir"*. *Encoder* takes a sequence of English words as input and encodes it with a fixed sized vector of numbers. *Decoder* takes this vector as input and produces the sequence of French words.

Recently [21] argued that abstractive summarization could be seen as a translation problem from text to summary. Automated text translation is a very prolific research domain. Among the numerous approaches, we wanted one that made as little assumption as possible on the problem to solve. We chose a neural network approach that maps an input sequence (the document) to an output sequence (the summary) and is commonly used for neural machine translation. It can also be applied to the summarization problem additional knowledge about the nature of the sequences other than the list of words they are made of. Specifically, we use a *sequence to sequence recurrent neural network with GRU cells and attention-based decoder* [23, 7, 13]. The fact that the neural network is *recurrent* allows one to have an input sequence of any length. In Figure 1, it means that each cell (green or yellow boxes) is actually the same in the network that links to itself. The figure itself shows an hypothetical unrolled network where each word is processed by one cell. The *Sequence to sequence* neural network (aka encoder decoder neural network) means that we join two recurrent neural networks, where the first learns to encode the input sequence into a fixed sized vector (see Figure) and the second learns to decode that vector into an output sequence. Notice that this encoded vector is actually the (hidden) state of the cell at the end of the input sequence. In the encoder, the output of the cell (vertical arrow) is ignored, thus not represented in the Figure. In the decoder, the output is plugged back as the input for the next step. The decoder generates the output sequence word by word including the special "end of sequence" word (<s> in the Figure). This way, the decoder decides by itself the length of the output sequence. The fact that both are recurrent neural networks, means that the size of the input and output sequences can be decorrelated which is important for the purpose of summarization. Finally such networks tend to give more importance to the latest words and possibly forget the first ones. This is an issue for us because, if on average, English sentences have around 20 words, in our experiments, the average size of methods exceeds 130 words [26]. The common approach to fight against this problem is the *attention mechanism* (not illustrated in the Figure) that ensures that the position of a word in the input sequence does not affect the output. It takes the form of an additional layer that decides independently what attention to give to each input word.

In Figure 1, we give two examples of the process, on top, of the translation from English to French, and on bottom, of the translation of a tokenized source code sentence into a tokenized method name.

3 Applying the Learning Models to Source Code

In this section, we first give some specificities of the Pharo language that influence how we applied the model. Then we describe how to prepare the methods to train and apply the models described in the previous section.

Each method translates into two sequences of words:

- The words extracted by splitting all identifiers in the method's body. This will form a "document" that the models must summarize.
- The words extracted by splitting the method's name. This will be used to train the models on body summarization.

3.1 Pharo Language Specificities

As stated in the Introduction, the language syntax and conventions have an impact on how we can apply the model. Pharo[4] has a number of specificities that illustrate this point:

1. Variables are not statically typed, a model could learn from the information about the types of variables, but it is not available in Pharo.
2. Pharo has in-place argument in method names. Unlike C like language where arguments are grouped in parentheses at the end of a function call, in Pharo, arguments are inserted between the parts of a method name. This way, for example, the Java statement "`bob.send(email,emma);`" translates to "`bob send: email to: emma.`" in Pharo. The consequences are that method names are longer, their vocabulary larger and they must be split at the right place to introduce the parameters. Longer names are not a problem *per se*, this is part of the model training process to learn the right length for a name. Splitting names at the correct place to introduce parameter is out of the scope of this paper. We generate method names as a list of words without the colons.

[4] pharo.org

3. Because of the in-place arguments, method names in Pharo contain many stop words such as *on*, *with*, *and*, *to*, etc. Method names in languages like Java, Python, or C contain mostly nouns, verbs, and adjectives that are highly representative of method's purpose, for example, `print(string, stream)` or `add(element, array)`. Method names in Pharo use stop words to separate and describe those arguments, for example, `print: string on: stream` or `add: element to: array`. It can be harder for extractive model to generate stop words because they do not necessarily appear in source code, and will most likely be discarded as too generic. We will explore the issue of stop words in Section 5.1.
4. Programming conventions dictate that methods should be very short in Pharo. In our experiment, the median length was three lines of code (mean around six) [26]. This means that documents are short with few words which may influence the models.
5. Some meta information about the methods is specified by calling certain methods from the body of the given method. For example an abstract method is a method that calls `self subclassResponsibility`. This obviously affects the model because the body of an abstract method has no relation with its name.

3.2 Data Preparation: Extracting Words from Source Code

To extract words from the methods body we: (1) only keep the identifiers in the method's body; (2) split the identifiers into alphabetic words; (3) convert words to lowercase. Only keeping identifiers means that we remove comments, symbols, numbers, strings, and characters. Note that in Pharo, `true`, `false`, and `nil` are not reserved words but variable containing predefined objects. As such we keep them as identifiers. Local variable declarations (see below, `|n|`) and block arguments (see below, `:char`) are also ignored. Because there is no type associated to them and they, normally, appear elsewhere in the source code, they bring very little information. After splitting the identifiers, we may end up with numbers that were part of the identifier, these are discarded as they are not alphabetical words (we also ignore the underscores and/or colons that were parts of the identifiers). So for example, the following hypothetical method body that computes the length of a string[5]:

```
"Computes the length of aString"
| n |
n := 1.
self do: [:char | n := n + 1].
↑ n
```

will result in the "document": *"n self do n n n"*. This is a perfect example of the issue that short methods raise. It is impossible to abstract a correct name for this method from the sequence of words we extract. Fortunately, many methods are

[5] The actual meaning of the code is not important, but, double quotes delimit comments, pipes delimit local variables declaration, square brackets delimit lambda functions, and caret is a return

more informative. Method names are decomposed in the same way. For example, the name printOn: delimiter : last : becomes: "*print on delimiter last*"

3.3 Data Preparation: Filtering the Dataset

Furthermore, to apply the model, some methods need to be discarded:

- Methods with names from which no words could be extracted. Overloaded operators (*e.g.* +) or strangely named methods (*e.g.* _42) would produce no words at the tokenization step described in Section 3.2;
- Methods with empty body after preparation. Similarly, some very short methods, for example returning only a constant, or empty hooks, would have an empty body after preparation;
- Fully duplicated methods, with same name and implementation, are reduced to only one instance so as not to bias the model. Note that duplicated method names with different bodies (*e.g.* toString in Java) are kept;
- Too long methods, more than 500 words in the body, are rejected for practical reasons. Training the model with such methods becomes too long to be practical. This problem comes from the attention mechanism (see Section 2.2) that requires to know the maximum input length and becomes very slow because of that, even for the shorter methods;
- Getters and setters are very easy to generate (most IDEs can actually do it), yet probabilistic models could fail on them. Therefore it seems better to leave them out of the scope of a method name generation model;
- Test methods name can also be easily generated, but this is done from a completely different source of information, usually the name of a method or class that is being tested, not their own body. Test method naming also follows different conventions that would require a specific model to learn;
- We said that abstract methods in Pharo were implemented with only a call to subclassResponsibility . It makes sense to remove them from the training set as it is impossible to learn anything from their body and this would just introduce noise in the model. The same goes for methods consisting of the sole call to shouldNotImplement call, that allows to "remove" an unwanted inherited method.
- Methods whose body would consist only of the words super, self[6], true, false , nil are also filtered out. Such bodies only add noise and are akin to empty bodies.

3.4 Finetuning the Models

Training the probabilistic models involves hyperparameter tuning. This is done by training them on a (large) subset of the dataset and validating them on a disjoint (smaller) subset. By fine tuning the parameters, one tries to achieve the best possible results. Assuming the dataset used is large and representative

[6] *this*

enough, this needs to be done only once for a given programming language. For training set, we used 70% of the whole dataset, and for validating, we used another 10%. The remaining 20% were required for the study and comparison of the two models and will be discussed in the next section. The Extractive model (TF-IDF), unlike the abstractive one, cannot automatically decide how many words it should generate. Therefore we used the following heuristics: We keep only those words with TF-IDF score above a certain threshold. There is a lower and upper boundaries on the number of words that can be kept. If no words pass the threshold, we keep the one with the highest TF-IDF score. If too many words pass the threshold, we keep only the highest TF-IDF scores. The TF-IDF threshold was fixed to the one that gave the best F1 score (see Section 4.1) on the validation dataset. The value is 2.5 for Pharo. The upper bound value was set to 5 words, so that the n-gram model would not take too much time to find the most meaningful order of those words. For the Abstractive model (Sequence to Sequence Neural Network) one needs to tune parameters such as the size of the hidden state vector, the learning rate, and the teacher forcing ratio[7]. These parameters were selected to give the highest F1 score on the validation dataset. The quality measures are discussed in Section 4.1). For Pharo, we recommend to set hidden state vector=256 ; learning rate=0.01; and teacher forcing ratio=0.5.

4 Experiment Setup

To compare the two models, we set up an experiment on some real world Pharo projects. We shuffled and split our data into three non-intersecting subsets. The first two were already presented in the previous section: training set (70%) and validation set (10%). For comparing the models, we also need a third independent set — test set (20%) — it is used to evaluate the final trained model on data not seen during the training itself so that the training and parameter tunning are not biased towards the test.

4.1 Quality Metrics

A given method name can be considered good by one developer and bad by another. In this study, we adopt a simplified approach for automatic evaluation which assumes that most methods in our dataset are well named, and therefore can be used as ground truth to evaluate our models. The actual name is called *reference name*, the ones generated by the models are the *candidate names*. We report four different metrics of similarity between candidate and reference names: *exact match*, *average precision*, *average recall*, and *average F1 score*. *Exact match score* is the simplest and the strictest metric. It is the percentage of candidate names that match exactly the reference names. This is our only metric that takes into account the order of words. Exact match score is easy to interpret,

[7] The probability that during training the word generated by the model is substituted by the word from a real name. It is used to make the training smoother

but very restrictive. Candidate name that is similar to the reference but does not match it exactly, will receive the score of 0, as if it was completely different. We used precision, recall, and their derived metric F1. These three metrics consider every name as a set of words. *Precision* counts the percentage of words from the candidate name that also appear in a reference name. *Recall* counts the percentage of words from the reference name that also appear in a candidate name. *F1 Score* is the harmonic mean[8] of precision and recall [22]. We compute these three metrics for each method and report the average of those values.

4.2 Random baseline

Because of the limited vocabulary [26] and the fact that source code is highly repetitive [12], we can get good results just by selecting the words randomly. Therefore we will also compare our models to a random model as a baseline: The *random extractive model* generates name for a given method by selecting K random words from its source code. A random abstractive model, selecting K random words from all method names in our training set, would make little sense as it would have close to 0% chance of finding the right word (in our dataset, there are 8,211 words [26]). We set K equal to 3, which is the average number of words in the method names from out training set.

5 Results

In this section, we present and discuss the results of our experiment evaluation. We experimented on 50 projects selected from Pharo ecosystem. The list of projects, some information about them, and how they were selected, is available in our technical report [26]. We collected 132,046 methods out of which we kept 92,127 (61%) after the filtering process described in Section 3. The three datasets described in Section 4 have: 64,488 methods in the training set; 9,212 methods in the validation set; and 18,425 methods in the test set. Table 1 contains several examples of method names that were generated by our abstractive model.

5.1 Evaluation Results

We present the results of the experiment in Table 2. As could be expected, the random model gives bad results for exact match (0%). Its results for precision (20%), recall (26%), and F1 score (21%) are not so bad. This is caused by the small vocabulary.

The extractive model shows an improvement over these results, with 2% exact match, precision of 28%, recall of 45%, and F1 score of 33%. The extractive model cannot propose new words that did not appear in the method body, this should reflect on a low recall which is not the case. Many methods in Pharo call

[8] Harmonic mean is more intuitive than the arithmetic mean when computing a mean of ratios

Table 1. Examples of method names generated with the abstractive model. The first column contains the source code of a method which was used as input, the second column contains the real method name which was unknown to the model, and the third column contains the generated name.

Source Code	Real Name	Generated Name
`self assert: self newNode isComment.`	testIsComment	testIsComment

`r := aColor red. g := aColor green.`
`b := aColor blue.`
	color	color

`aVisitor visitDraggableInteraction: self`
`with: args`
	acceptWith	accept

`aPackage isPackage ifFalse: [ ^ self ].`
`self addElement: aPackage`
`in: self packages.`
	addPackage	addPackage

Table 2. Evaluation results calculated on the test set for three models: the *random model* that selects three random words from source code, *extractive model* based on TF-IDF and n-gram model, and the *abstractive model* based on a sequence to sequence deep neural network.

Model	Exact Match	Precision	Recall	F1 Score
Random	0%	20%	26%	21%
Extractive	2%	28%	45%	33%
Abstractive	11%	46%	32%	36%

another one with a similar name (to add a default parameter for example) which could be an explanation here. Further studies are needed to better understand this issue. The abstractive model has the best results, with 11% exact match, precision of 46%, recall of 32%, and F1 score of 36%. The high exact match is surprising and may be as good as what a human would achieve. Additionally, in Figure 2, we plot the intermediary results of the abstractive model, one data point every 1,000 iterations. The scores are evaluated against the validation set, not the test set. This is what we used to finetune the parameters of the model. For comparison, we also draw the performance of the extractive model (dashed lines) evaluated on the validation set. F1 scores of the extractive and abstractive models are almost the same. However, extractive model performs worse based on the exact match score. We can try to explain this by the presence of many stop words (e.g. with, on, to) in the names of Pharo methods (see Section 3.1). One might argue that a language like Java, that usually does not exhibit such stop words in method names, could have better score here. To validate this hypothesis, we have identified 127 generic words such as *on*, *with*, *and*, etc. that are considered stop words in English[9]. We removed every occurrence of those words in the method names of training, validation, and test sets. Then we rerun the experiments and evaluated our three models on the new data to observe

[9] The complete list of stop words that we used in this study can be found here: https://gist.github.com/olekscode/125804150f2a559a171bf695c0a3f809

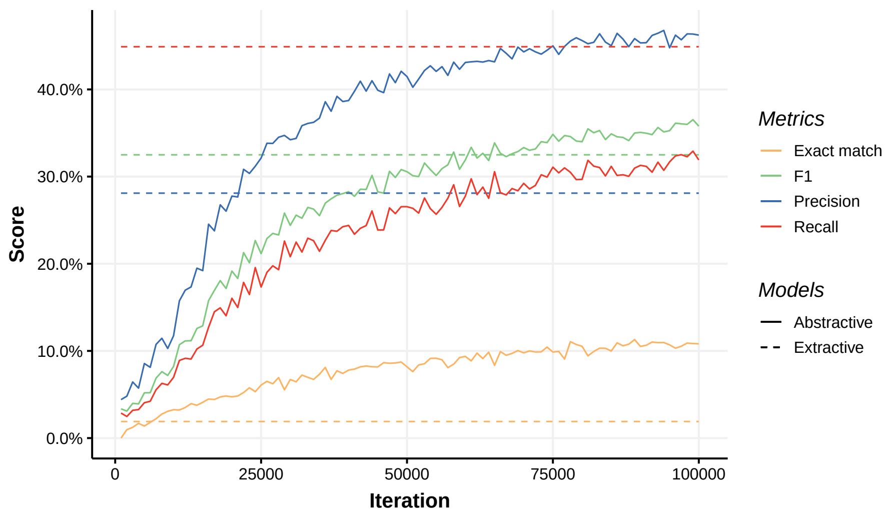

Fig. 2. Training of the abstractive model (measurements were taken once every 1000 iterations). The dashed lines are the scores achieved by the extractive model

the effect that stop words have on the generation of method names. Against our hypothesis, the exact match of the extractive model was not affected by removing the stop words, it remained 2%. As for the abstractive model, its exact match score increased by 2% giving us 13% of exactly matched method names. The changes of precision, recall, and F1 score for random, extractive, and abstractive models are inconclusive and seem to be purely mechanical.

6 Related work

Following the work of Gabel et al.[10] who performed the first study of the uniqueness of code and found that source code is highly repetitive, Hindle et al. [12, 11] explored the predictability of code, and claimed that source code is even more repetitive and predictable than natural languages. They claimed that this predictability allows us to model code with statistical language models, proposed the notion of software naturalness and pioneered the applications of natural language processing (NLP) to the source code. Deep learning proved to be very effective in modelling source code — in recent years, deep learning models for source code found many applications for code completion [24, 20, 2]. Bavi et al. [5] used auto-encoder network together with a recurrent neural network to reverse the minification of JavaScript and generate names for local variables. Allamanis et al. [1] introduced the first neural probabilistic language model for source code and used it to suggest method names. This model required a large set of hard-coded features, such as features from the containing class and the method signature. In the later study, Allamanis et. al. [3] proposed an end-to-end (meaning that it does not require manual feature selection) convolutional neural network with attention for method name generation. Alon et al. [4] also attempted to predict method names form their bodies by representing source code as a collection of paths in its abstract syntax tree and aggregating these paths into a single fixed-length code vector. Iyer et al. [14] proposed an LSTM network with attention to generate sentences that describe C# code snippets and SQL queries. Their model was trained to translate between the titles of questions posted on StackOverflow and code snippets from the accepted answers.

7 Conclusion

In this work, we explored and compared two machine learning models for text summarization when applied to the problem of generating descriptive method names from method bodies and thus improving the readability of source code. The first model is based on TF-IDF and n-gram language model, it performs the extractive text summarization by selecting important words from the source code of a method and putting them into a meaningful order. Second model is an attention-based sequence to sequence neural network which performs an abstractive summarization — it can generate method names from words that have never appeared in source code. After applying and evaluating our models on the dataset of methods collected from 50 real-life projects written in Pharo, we have reported the average precision score of 28% for extractive and 46% for abstractive models, as well as the average recall of 45% for extractive and 32% for abstractive model. 11% of method names generated by our abstractive model for methods from an independent test set are exactly the same as the real names given to those methods by developers.

Threats to validity The method names generated with our abstractive approach are only as good as the names on which the model was trained. The project that we have included into our dataset were handpicked by experts as the ones that follow good coding practices. However, we did not manually validate each one of the 64,488 method names in the training set, so this can be a threat to validity. A similar threat is the validity of evaluation. We have considered the real method names from our dataset as ground truth and used them to evaluate the generated names. Such approach is based on the assumption that the original names are good. In the follow-up study, we plan to perform a manual qualitative evaluation of the generated names.

Future work The evaluation technique could be enhanced by supporting the four automatic metrics of exact match, precision, recall, and F1 score with a human evaluation performed on a small subset of methods. For example, a model that generates a name, “on” for a method whose real name is “printOn:“, will be awarded with 100% precision and 50% recall. Alternatively, if the real method name is “sumOfIntegers”, a reasonably good name such as “addAllIntegerNumbers” will be scored with 0 by all metrics. Those cases would be easily spotted by a human evaluator. The same experiment should be tried with other programming languages as we saw that Pharo methods are typically small (a few lines of code) which limits the vocabulary available for both approaches. This can have good or adverse consequences on the results. In this work, we removed code comments and string literals because our study was focused on generating method names by summarizing source code. However, as we discussed in Section 3.2, many methods are very short and do not contain enough valuable information in their source code to generate a meaningful method name. A good extension for our study would be to utilize the natural language method descriptions provided in code comments. As it was mentioned in Section 1 where we discussed the motivation, the automatic suggestion of method names can be used to improve the readability of source code, which eventually would improve bug fixing and feature request incorporation times. We plan to target this problem through a controlled experiment or a longitudinal case study in the follow-up journal paper. Another interesting follow-up study would be to do cross project (or cross domain) training. In this paper, we trained on all projects (domains) mixed but it seems reasonable to assume that different projects would have different naming convention and vocabulary. Again, this could impact the results.

Acknowledgements

This work is based on the Master’s thesis of Oleksandr Zaitsev defended at the Ukrainian Catholic University [25]. Oleksandr would like to thank the University of Chile, Inria Lille, Pharo Association, and Arolla for financial support. Alexandre Bergel thanks the financial sponsor of Lam Research and project FONDECYT Regular 1200067.

References

1. Allamanis, M., Barr, E.T., Bird, C., Sutton, C.: Suggesting accurate method and class names. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 38–49. ACM (2015)
2. Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C.: A survey of machine learning for big code and naturalness. ACM Computing Surveys (CSUR) **51**(4), 81 (2018)
3. Allamanis, M., Peng, H., Sutton, C.: A convolutional attention network for extreme summarization of source code. In: International Conference on Machine Learning. pp. 2091–2100 (2016)
4. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: Learning distributed representations of code. arXiv preprint arXiv:1803.09473 (2018)
5. Bavishi, R., Pradel, M., Sen, K.: Context2name: A deep learning-based approach to infer natural variable names from usage contexts. arXiv preprint arXiv:1809.05193 (2018)
6. Beck, K.: Test Driven Development: By Example. Addison-Wesley Longman (2002)

7. Cho, K., Van Merriënboer, B., Bahdanau, D., Bengio, Y.: On the properties of neural machine translation: Encoder-decoder approaches. arXiv preprint arXiv:1409.1259 (2014)
8. Demeyer, S., Ducasse, S., Nierstrasz, O.: Object-Oriented Reengineering Patterns. Morgan Kaufmann (2002)
9. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison Wesley (1999)
10. Gabel, M., Su, Z.: A study of the uniqueness of source code. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. pp. 147–156. ACM (2010)
11. Hindle, A., Barr, E.T., Gabel, M., Su, Z., Devanbu, P.: On the naturalness of software. Communications of the ACM **59**(5), 122–131 (2016)
12. Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P.: On the naturalness of software. In: Software Engineering (ICSE), 2012 34th International Conference on. pp. 837–847. IEEE (2012)
13. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation **9**(8), 1735–1780 (1997)
14. Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). vol. 1, pp. 2073–2083 (2016)
15. Knuth, D.E.: Literate programming. The Computer Journal **27**(2), 97–111 (1984)
16. Koenig, A.: Patterns and antipatterns. Journal of Object-Oriented Programming (Mar 1995)
17. Lehman, M., Belady, L.: Program Evolution: Processes of Software Change. London Academic Press, London (1985), ftp://ftp.umh.ac.be/pub/ftp_infofs/1985/ProgramEvolution.pdf
18. Martin, R.C.: Clean code: a handbook of agile software craftsmanship. Pearson Education (2009)
19. Ramos, J.: Using tf-idf to determine word relevance in document queries. In: Proceedings of the first instructional conference on machine learning. vol. 242, pp. 133–142 (2003)
20. Raychev, V., Vechev, M., Yahav, E.: Code completion with statistical language models. In: Acm Sigplan Notices. vol. 49, pp. 419–428. ACM (2014)
21. Rush, A.M., Harvard, S., Chopra, S., Weston, J.: A neural attention model for sentence summarization. In: ACLWeb. Proceedings of the 2015 conference on empirical methods in natural language processing (2017)
22. Sasaki, Y., et al.: The truth of the f-measure. Teach Tutor mater **1**(5), 1–5 (2007)
23. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in neural information processing systems. pp. 3104–3112 (2014)
24. White, M., Vendome, C., Linares-Vásquez, M., Poshyvanyk, D.: Toward deep learning software repositories. In: Proceedings of the 12th Working Conference on Mining Software Repositories. pp. 334–345. IEEE Press (2015)
25. Zaitsev, O.: Aspects of software naturalness through the generation of identifier names. Master's thesis, Ukrainian Catholic University, Faculty of Applied Sciences, Department of Computer Sciences, Lviv, Ukraine (Jan 2019), http://er.ucu.edu.ua/handle/1/1338, under sup. of Stéphane Ducasse and Alexandre Bergel
26. Zaitsev, O., Ducasse, S., Anquetil, N.: Characterizing pharo code: A technical report. Technical report, Inria Lille Nord Europe - Laboratoire CRIStAL - Université de Lille ; Arolla (jan 2020), https://hal.inria.fr/hal-02440055