Paper ML09

AI assisted code reviews

Igor Khorlo, Syneos Health

ABSTRACT

Code reviews become a very crucial and essential task in quality control assurance in the pharmaceutical industry. Some companies implemented code reviews as a standard validation procedure for simple SAS programs instead of double programming. In this paper, we will consider how Artificial Intelligence can assist with this process by building a model that calculates an entropy for each token in the SAS program. Not going deep into details of what is the definition of entropy, we are going to rephrase it in simple words – a measure of suspiciousness that the thing is incorrect. We are going to draw a heatmap for a given piece of SAS source code, that will highlight areas of the high entropy, that we, for instance, can use during the code reviews to highlight the areas that should be reviewed first.

INTRODUCTION

Artificial Intelligence (AI) become a buzzword in the computational science fields. Deep Learning (DL), a subfield of AI, have been successfully applied in various fields including computer vision, speech recognition, natural language processing, audio recognition, machine translation, medical image analysis, where they have produced results comparable to and in some cases superior to human experts. Sequence models, in particular, have been successfully applied to natural language processing (NLP) tasks. A recent example of GTP-2[1] and similar models shows incredible performance in reading comprehension, machine translation, question answering, and summarization. Authors even decided not to release the weights of GTP-2 due to the concerns about malicious applications of the technology[2].

Besides, Machine Learning (ML) on source code gets a lot of attention in software development circles. As it turned out, applying the same models from NLP tasks to source code produces very promising results and could solve some problems arising in software development. On the other hand, large corpora of open source software projects made it possible to train these machine learning models. Such models have enabled research on a broad suite of new software engineering tools. Examples of such are:

- new tools for autocompletion[3][4]
- program repair[5]
- suggesting readable function and class names[6]
- summarizing source code, word2vec style embeddings[7][8][9][10]
- predicting bugs[11]
- detecting code clones[12]
- comment generation[13]
- variable de-obfuscation[14], identifier misuse[15].

[1] Article about GTP-2 from Open AI – https://openai.com/blog/better-language-models/
[2] GPT-2: 6-Month Follow-Up – https://openai.com/blog/gpt-2-6-month-follow-up/
[3] Raychev u.a. (2014)
[4] Alona Bulana, Igor Khorlo (2019)
[5] Ray u.a. (2016)
[6] Allamanis u.a. (2015)
[7] Allamanis u.a. (2016)
[8] Iyer u.a. (2016)
[9] Mikolov u.a. (2013)
[10] Alon u.a. (2019)
[11] Pradel; Sen (2018)
[12] White u.a. (2016)
[13] Hu u.a. (2018)
[14] Bavishi u.a. (2018)
[15] Allamanis u.a. (2017)

Code review is a software quality assurance (QA) activity in which one or several persons check a program mainly by viewing and reading parts of its source code, and they do so after implementation or as an interruption of implementation. At least one of the persons must not be the code's author. The humans performing the checking, excluding the author, are called "reviewers". Many companies implemented code review as a standard QA procedure, e.g. final review from the team lead before merging into master. Google published a document with recommendations for How to do a code review. PhUSE has a wiki section dedicated to Code Review. Some companies even implemented code reviews as a standard validation procedure for simple SAS programs instead of double programming.

In this paper, we will consider how deep learning can assist with the process of code review by building a piece of software that will use sequence model that calculates an entropy for each token in the SAS program. Not going deep into details of what is the definition of entropy, we will be talking about this later in the paper, we are going to rephrase it in simple words – a measure of suspiciousness that the thing is incorrect, meaning that we cannot predict it so easily with our model. We are going to draw a heatmap for a given piece of SAS source code, that will highlight areas of the high entropy, that we, for instance, can use during the code reviews to pick out the areas that should be reviewed first. It doesn't mean that the highlighted areas are incorrect, it just means that the probability of having these statements in this program is very low.

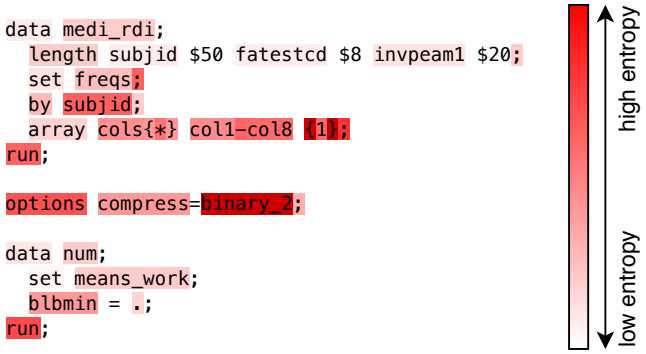

Figure 1: Entropy heatmap for a sample SAS program. Background colour represents the entropy of a token. High entropy (red) means suspicious – low probability, low entropy (white) means a token is regular – high degree of confidence. As we can see the highest areas of entropy are indeed an incorrect SAS code – `binary_2` value does not exist for COMPRESS= System Option, and the way how the array `cols` is initialised is not a correct SAS syntax.

TOY MODEL NEURAL NETWORK

To give you motivation and understanding of how this works we start from a simplified case. We will use a fully connected neural network, will consider only the DATA Step code and will limit our vocabulary to around 57 words (tokens) which we will encode as one-hot vectors. Tokenization is a well-established process for structured languages parsing. We are going to use SAS Parser from SASLint project[16] to tokenize SAS source code corpus. The following is an example of a SAS program tokenization:

```
data mess2;                              | DATA <ID> ;
  merge mess ae2;                        | MERGE <ID> <ID> ;
  by usubjid;                            | BY <ID> ;
  set ads.adlb(                          | SET <ID> . <ID> (
    where=(paramcd='BACT'                | WHERE = ( <ID> = <STR>
           and visit = 'Screening'))     | OP_AND <ID> = <STR> ) )
    key=keyvar / unique;                 | KEY = <ID> / <UNK> ;
  data = 123;                            | <ID> = <INT> ;
run;                                     | RUN ;
```

[16] See details in Khorlo (2018)

Since we want to limit our vocabulary as much as possible, we will encode rarely faced tokens as <UNK>, all identifiers such as variables, datasets, libraries or formats as <ID>, string or number literals as <STR> or <NUM> correspondingly. Each token is encoded as a one-hot vector. For instance, the above tokens will be encoded as following:

$$\texttt{DATA} = (1, 0, ..., 0) \in \mathbb{R}^{57}$$
$$\texttt{;} = (1, 0, ..., 0) \in \mathbb{R}^{57}$$
$$\texttt{<ID>} = (0, 1, ..., 0) \in \mathbb{R}^{57}$$
$$\texttt{MERGE} = (0, 1, ..., 0) \in \mathbb{R}^{57}$$
$$...$$
$$\texttt{RUN} = (0, 0, ..., 1) \in \mathbb{R}^{57}$$

The next thing we are going to introduce is the context. The context in this model means **the sequence length** we consider for the model input. Since it's a fully connected neural net – the input is fixed size. The following diagram shows the architecture of the network:

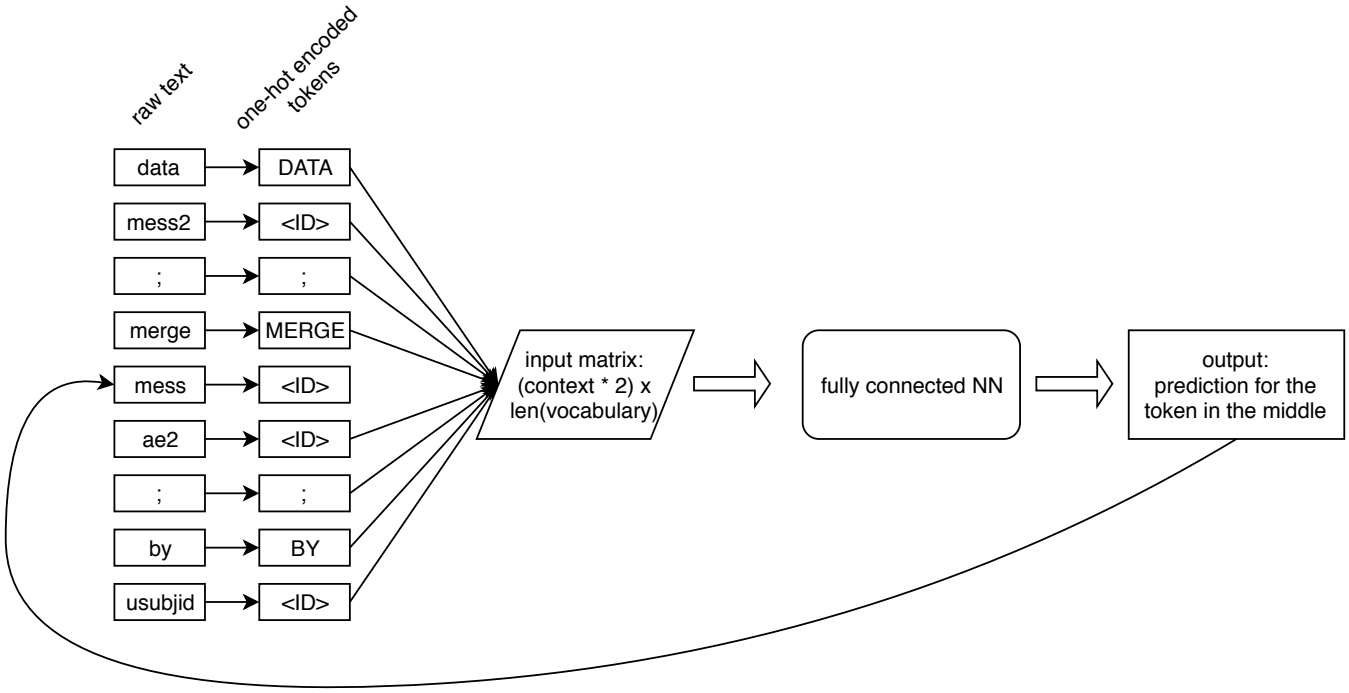

Figure 2: Raw program text is tokenized and one-hot encoded. Then `context*2 + 1` tokens are taken for a single iteration. Token in the middle is removed from the input since this is the token we are going to predict. Input matrix's size is `(context*2)x (vocabulary size)`. Fully connected model is defined in the way that a dense layer with softmax activation is the output layer – this will give us a vector in $\mathbb{R}^{57}$ space and components in `[0, 1]` interval. Therefore, we can treat it as a probability distribution for the token in the middle.

```
# number of tokens before/after as context
context = 10

# preprocess the input, define train and test sets
# ...

# define model
model = Sequential([
    Dense(100, activation='relu', input_shape=(context*2, len(vocab))),
    Dense(50, activation='relu'),
```

```
    Flatten(),
    Dense(len(vocab), activation='softmax')
])
model.compile(Adam(), loss='categorical_crossentropy', metrics=['accuracy'])

# train
model.fit(train_in, train_out, epochs=10)
```

Model summary:

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 20, 100)           5800
_________________________________________________________________
dense_2 (Dense)              (None, 20, 50)            5050
_________________________________________________________________
flatten_1 (Flatten)          (None, 1000)              0
_________________________________________________________________
dense_3 (Dense)              (None, 57)                57057
=================================================================
Total params: 67,907
Trainable params: 67,907
Non-trainable params: 0
```

After training this network you could get to around 60-70% in performance because the network itself is too small (only 67k parameters) to accumulate so many dependencies in SAS code. Also, the context size is limited and this dramatically influences the performance.

The following code evaluates the trained model on the sample input and prints a probability distribution for the predicted token, particularly for MERGE:

```
# evaluate
seq = encode('data mess2; merge mess ae2; by usubjid; set ads.adlb(')
x = select_input(seq, 4)
p = model.predict(x) # p.shape = (1, 57)

# print 5 highest probable tokens
for times in range(5):
    i = np.argmax(p) # select highest probability element from p
    print(code_to_token[], p[0][i])
    p[0][i] = 0

# MERGE 0.7913401
# SET 0.1522897
# FORMAT 0.013148197
# INFORMAT 0.00796120195
# FILE 0.00093120028
```

The interesting thing here is that MERGE token is predicted very the high degree of confidence. Admittedly, it is quite expected that there is a `merge` statement at that place. What we are going to do next is instead of using these probabilities to predict things, we are going to use the rest of the predictions to detect anomalies. Rather than being interested in the highest probability, we are going to answer the questions – What is the probability that this token is the right one at this place? Or how probable is that this is the *right* thing? For example, imagine, if instead of the MERGE token we would have a LENGTH token at that place. The probability of having LENGTH is

```
print(p[0][token_to_char('LENGTH')])
# 0.00048275382
```

which is very low probability.

ENTROPY

To sum this up, we have a neural network, and it generates predictions for each token in the SAS program. And instead of using these predictions to predict things (like building an autocompletion engine[17]), we are going to use **the rest** of the predictions to find irregular things (issues). This measure of irregularity we will call entropy. Not going deep into details of what is the definition of entropy, we are going to rephrase it in simple words – a measure of suspiciousness that the thing is incorrect. When we get to the high enough level of entropy for a particular token, we can mark that place saying – this is suspicious, we cannot predict that so easily.

$$entropy = -\ln \hat{y}$$

where $\hat{y}$ – the probability of the token, ln – natural logarithm. By the way, high entropy does not mean that something is wrong, it means that this is unusual – you should definitely take a look at those places in that piece of code.

FURTHER MODEL IMPROVEMENTS

With the approach from the previous section, you won't get so far in performance. And one of the main reasons for this is a very limited context, i.e. taking 10 tokens before and after is just not enough for the model to learn long-term dependencies, which are common in a program source code, for example:

- open/close quotes (that can expand to quite long sequences)
- step boundaries like `data/run;`
- fundamental `do;/end;` blocks.

To address this issue you could try to use more sophisticated sequence models such as recurrent neural networks (RNN). However, vanilla RNNs still won't work so well because of the vanishing gradient problem and are poor in capturing long term dependencies[18]. To address that issue you could try to use an RNN modification called a Long Short-Term Memory (LSTM)[19] network or its simplification a Gated Recurrent Unit (GRU)[20] network. Both networks work better in practice and deliver better results than RNNs[21]. However, this is still not enough, and recent studies and results show that the top model you could use for such purpose is a Transformer model[22]. This architecture was used in many recent successful NLP models such as GPT-2, BERT, XLNet, and others. Besides, it was successfully applied in the commercial product for source code autocompletion TabNine[23], which is a very promising sign.

CONCLUSION

In this paper we shown how to build a deep neural network and obtain an entropy for each token. This entropy could be later used to assist code review by highlighting the areas that should be reviewed first. Furthermore, this tool could be integrated into Continuous Integration process you use at your company, e.g. heatmap generation could be triggered for each pull request in GitLab as assisting process during the code review.

On the other hand, tokenization of SAS programs and dealing with arising problems like identifier representation, string literals understanding, SAS macro code are still the area where improvement is possible.

[17] See Alona Bulana, Igor Khorlo (2019)
[18] See Bengio u.a. (1994)
[19] See Hochreiter; Schmidhuber (1997); and Olah (2015)
[20] See Cho u.a. (2014)
[21] See Chung u.a. (2014)
[22] See Vaswani u.a. (2017)
[23] TabNine is the all-language autocompleter – https://tabnine.com/

CONTACT INFORMATION

Igor Khorlo

Syneos Health

Berlin, Germany

igor.khorlo@syneoshealth.com

REFERENCES

Allamanis, Miltiadis; Barr, Earl T.; Bird, Christian; Sutton, Charles (2015): Suggesting Accurate Method and Class Names. New York, NY, USA: ACM, URL: http://doi.acm.org/10.1145/2786805.2786849 [Accessed: 3.10.2019]

Allamanis, Miltiadis; Brockschmidt, Marc; Khademi, Mahmoud (2017): Learning to Represent Programs with Graphs. In: *CoRR*, Band abs/1711.00740, 2017, URL: http://arxiv.org/abs/1711.00740 [Accessed: 3.10.2019]

Allamanis, Miltiadis; Peng, Hao; Sutton, Charles (2016): A convolutional attention network for extreme summarization of source code.

Alon, Uri; Zilberstein, Meital; Levy, Omer; Yahav, Eran (2019): Code2Vec: Learning Distributed Representations of Code. In: *Proc. ACM Program. Lang.*, Band 3, Ausgabe POPL, 1.201940:1–40:29, URL: http://doi.acm.org/10.1145/3290353 [Accessed: 3.10.2019]

Alona Bulana, Igor Khorlo (2019): A Neural Net Brain for an Autocompletion Engine: Improving the UX Through Machine Learning. In: *SAS® Global Forum 2019 Proceedings*, 2019, URL: https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2019/3200-2019.pdf [Accessed: 3.10.2019]

Bavishi, Rohan; Pradel, Michael; Sen, Koushik (2018): Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts. In: *CoRR*, Band abs/1809.05193, 2018, URL: http://arxiv.org/abs/1809.05193 [Accessed: 3.10.2019]

Bengio, Yoshua; Simard, Patrice; Frasconi, Paolo (1994): Learning Long-Term Dependencies with Gradient Descent is Difficult. In: *IEEE Transactions on Neural Networks*, Band 5, Ausgabe 2, 1994157–166, URL: http://www.iro.umontreal.ca/~lisa/pointeurs/ieeetrnn94.pdf [Accessed: 23.3.2019]

Cho, Kyunghyun; Merrienboer, Bart van; Bahdanau, Dzmitry; Bengio, Yoshua (2014): On the Properties of Neural Machine Translation: Encoder-Decoder Approaches., URL: https://arxiv.org/abs/1409.1259 [Accessed: 23.3.2019]

Chung, Junyoung; Gulcehre, Caglar; Cho, KyungHyun; Bengio, Yoshua (2014): Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling., URL: https://arxiv.org/abs/1412.3555 [Accessed: 23.3.2019]

Hochreiter, Sepp; Schmidhuber, Jürgen (1997): Long Short-Term Memory. In: *Neural Comput.*, Band 9, Ausgabe 8, 11.19971735–1780, URL: http://dx.doi.org/10.1162/neco.1997.9.8.1735 [Accessed: 23.3.2019]

Hu, Xing; Li, Ge; Xia, Xin; Lo, David; Jin, Zhi (2018): Deep Code Comment Generation. New York, NY, USA: ACM, URL: http://doi.acm.org/10.1145/3196321.3196334 [Accessed: 3.10.2019]

Iyer, Srinivasan; Konstas, Ioannis; Cheung, Alvin; Zettlemoyer, Luke (2016): Summarizing Source Code using a Neural Attention Model. Berlin, Germany: Association for Computational Linguistics

Khorlo, Igor (2018): SASLint: A SAS® Program Checker. In: *SAS® Global Forum 2018 Proceedings*, 2018, URL: https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/2543-2018.pdf [Accessed: 23.3.2019]

Mikolov, Tomas; Sutskever, Ilya; Chen, Kai; Corrado, Greg; Dean, Jeffrey (2013): Distributed Representations of Words and Phrases and Their Compositionality. USA: Curran Associates Inc., URL: http://dl.acm.org/citation.cfm?id=2999792.2999959 [Accessed: 3.10.2019]

Olah, Christopher (2015): Understanding LSTM Networks. 2015, URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs [Accessed: 23.3.2019]

Pradel, Michael; Sen, Koushik (2018): DeepBugs: A Learning Approach to Name-based Bug Detection. In: *Proc. ACM Program. Lang.*, Band 2, Ausgabe OOPSLA, 10.2018147:1–147:25, URL: http://doi.acm.org/10.1145/3276517 [Accessed: 3.10.2019]

Ray, Baishakhi; Hellendoorn, Vincent; Godhane, Saheel; Tu, Zhaopeng; Bacchelli, Alberto; Devanbu, Premkumar (2016): On the "Naturalness" of Buggy Code. New York, NY, USA: ACM, URL: http://doi.acm.org/10.1145/2884781.2884848 [Accessed: 3.10.2019]

Raychev, Veselin; Vechev, Martin; Yahav, Eran (2014): Code Completion with Statistical Language Models. In: *SIGPLAN Not.*, Band 49, Ausgabe 6, 6.2014419–428, URL: http://doi.acm.org/10.1145/2666356.2594321 [Accessed: 3.10.2019]

Vaswani, Ashish; Shazeer, Noam; Parmar, Niki; Uszkoreit, Jakob; Jones, Llion; Gomez, Aidan N.; Kaiser, Lukasz; Polosukhin, Illia (2017): Attention Is All You Need. In: *arXiv preprint arXiv:1706.03762v5*, 2017, URL: https://arxiv.org/abs/1706.03762v5 [Accessed: 3.10.2019]

White, Martin; Tufano, Michele; Vendome, Christopher; Poshyvanyk, Denys (2016): Deep Learning Code Fragments for Code Clone Detection. New York, NY, USA: ACM, URL: http://doi.acm.org/10.1145/2970276.2970326 [Accessed: 3.10.2019]