Paper 3200-2019

A Neural Net Brain for an Autocompletion Engine: Improving the UX Through Machine Learning

Alona Bulana, Independent Consultant   Igor Khorlo, Syneos Health™

ABSTRACT

Autocomplete feature has been present in IDEs and source code editors for more than a decade. It speeds up the process of coding by reducing typos and other common mistakes. SAS® Studio, for example, suggests the next appropriate keyword in a context. Snippets are also an essential part of every decent IDE. That software usually uses the declarative approach to implement autocompletion. Mainly, the problem with this approach is the size of an implementation which scales with a language's grammar size, leading to very complex logic.

In this paper, we solve the autocomplete engine construction problem using a machine learning way, which automatically derives an autocompletion engine from a language without explicitly being programmed. The idea is to use generative models based on Recurrent Neural Network (RNN) trained on a SAS source code to generate the suggestions, like what you are going to type next, much like your phone predicts your next words. An output of a generative model is a new piece of something that looks like SAS source code. We call it "looks like" because each output of a generative model is sort of unique and we cannot be sure that the text generated is a valid SAS source code. So here is where SASLint[1] comes to the rescue. SAS Parser from SASLint project will be used to check the validity of generated text that it is a valid SAS source code and suggest it to a user as an auto-completion. Autocompletion functionality will be demonstrated in the Web-based text editor build on the top of Ace[2] (demos – autocomplete[3], entropy heatmap[4]). It sounds like something unbelievable if you hear this first time, but it is really working, and this is impressive.

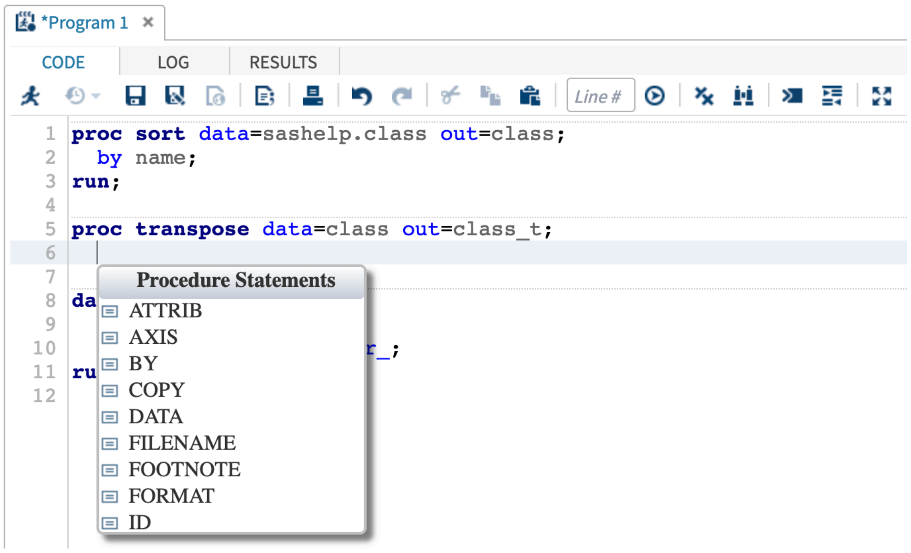
This application uses techniques of machine learning & neural networks, and it intends to improve the developer experience (so-called DX) of using SAS language.

INTRODUCTION

So what is an autocompletion engine in terms of source code? Autocompletion is a feature in which an application predicts the rest of input which a user intends to type. It dramatically speeds up human-computer interactions especially when the prediction is correct. It works best in areas with a limited number of possible statements, or when some statements are much more common, or writing structured and predictable text (as in source code editors). Autocompletion in programming languages (also known as code completion or code suggestion) is greatly simplified by the regular structure of the programming languages. There are usually only a limited number of words, statements, keywords meaningful in the current context, such as names of variables in the current DATA Step, statements in the PROC PRINT, or built-in functions. It can be extremely useful in a situation that you are typing some code from a SAS step (DATA or PROC) that you rarely work with, and you can recall yourself a forgotten option or statement name using the autocompletion.

[1]See Khorlo (2018)
[2]Ace is an embeddable code editor written in JavaScript – https://github.com/ajaxorg/ace
[3]https://saslint.com/rnn-autocomplete
[4]https://saslint.com/rnn-entropy

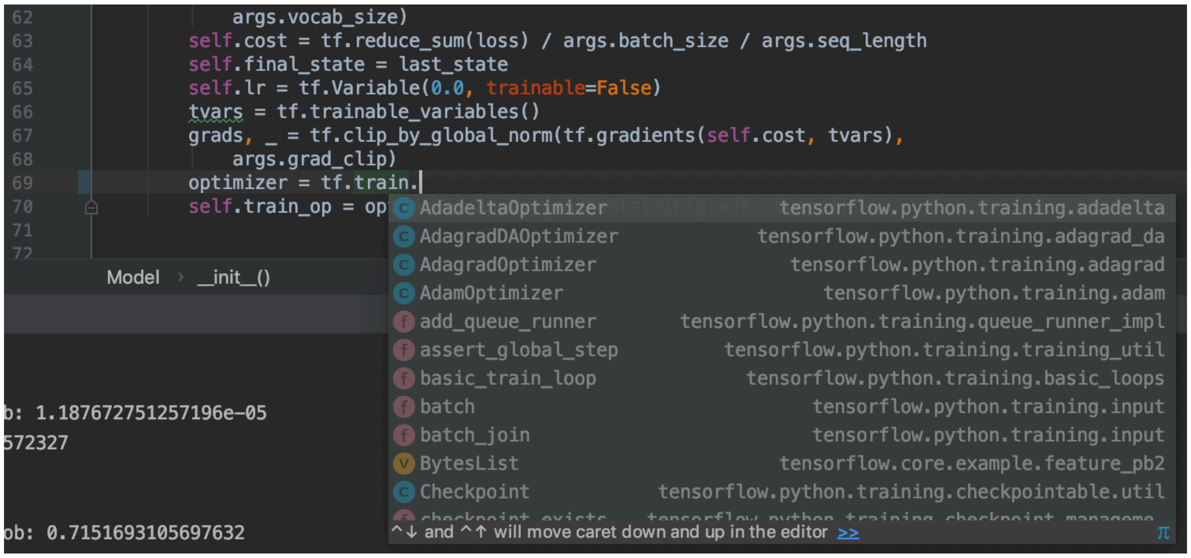
SAS Studio or SAS® Enterprise Guide® provide code completion in the context. The following example shows autocompletion for statements in the TRANSPOSE Procedure context:



```
proc sort data=sashelp.class out=class;
  by name;
run;

proc transpose data=class out=class_t;
  |

da
                                 r_;

ru
```

Figure 1: Code completion suggestions by SAS Studio 4.3 in the context of PROC TRANSPOSE

Another example is from the PyCharm IDE by JetBrains that is showing recommended completion items for your code context at the top of the completions list. The example below illustrates this in action for Tensorflow framework in Python code:



```
                args.vocab_size)
            self.cost = tf.reduce_sum(loss) / args.batch_size / args.seq_length
            self.final_state = last_state
            self.lr = tf.Variable(0.0, trainable=False)
            tvars = tf.trainable_variables()
            grads, _ = tf.clip_by_global_norm(tf.gradients(self.cost, tvars),
                args.grad_clip)
            optimizer = tf.train.|
            self.train_op = op
```

```
AdadeltaOptimizer          tensorflow.python.training.adadelta
AdagradDAOptimizer         tensorflow.python.training.adagrad_da
AdagradOptimizer           tensorflow.python.training.adagrad
AdamOptimizer              tensorflow.python.training.adam
add_queue_runner           tensorflow.python.training.queue_runner_impl
assert_global_step         tensorflow.python.training.training_util
basic_train_loop           tensorflow.python.training.basic_loops
batch                      tensorflow.python.training.input
batch_join                 tensorflow.python.training.input
BytesList                  tensorflow.core.example.feature_pb2
Checkpoint                 tensorflow.python.training.checkpointable.util
checkpoint_exists          tensorflow.python.training.checkpoint_manageme
^↓ and ^↑ will move caret down and up in the editor >>
```

```
b: 1.187672751257196e-05
572327

ob: 0.7151693105697632
```

Figure 2: Code completion suggestions by PyCharm 2018.1.4 Community Edition for the context of the external library(Tensorflow)

What unites all the autocompletion engines from above is the declarative approach in their implementation (disclaimer: at the moment of writing the paper – 23 March 2019). The advantage of this approach is that we know what code suggestions will be triggered in a specific context – that is how the declarative approach works. However, the disadvantage of this method is the size of an implementation which scales with a language's grammar size, leading to very complex logic, especially for such diverse grammars like SAS language has.

The method of implementing code completion we consider in this paper consists of applying a Natural Language Processing (NLP) techniques to a SAS source code. In particular, we will train several variations of Recurrent Neural Networks (RNN) on different kind of sequences – characters and tokens.

RECURRENT NEURAL NETWORKS

Depending on your background you might be curious why we picked an RNN type model for this problem? The most significant limitation of traditional Neural Networks (and Convolutional Neural Networks) is that they work with a fixed-sized input and produce a fixed-sized output. RNN architecture is devoid of this disadvantage and allows us to operate over sequences of arbitrary length. Sequences of vectors can be used as an input, output, or both:

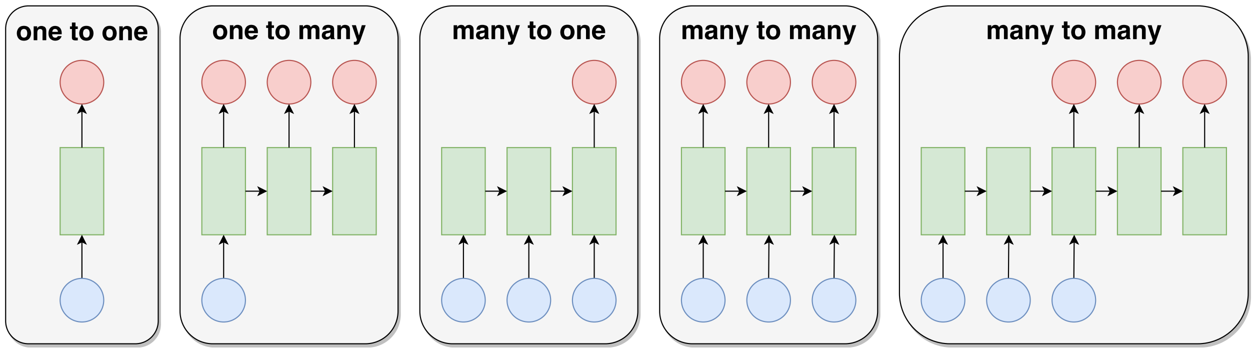


Figure 3: Input vectors are in blue, output vectors are in red and green vectors are a hidden layer that holds the RNN's state. From left to right: **(1)** Traditional NN without recurrence, from fixed-sized input to fixed-sized output (e.g. image classifier). **(2)** Fixed-sized input to sequence output (e.g. image captioning). **(3)** Sequence input to fixed-size output (e.g. sentiment of an input sentence). **(4)** Synced sequence input and output (e.g. real-time speech recognition). **(5)** Sequence input and sequence output (e.g. translation). Note that in 2, 3, 4, 5 cases have no restrictions on the sequences' lengths.

One more thing that we would like to mention is that the standard RNN units are suffering from the vanishing gradient problem and are poor in capturing long term dependencies[5]. This weakness makes it hard for RNNs to deal with long-range dependencies, which are common in program source code such as open/close quotes (that can expand to quite long sequences), step boundaries like `data/run;`, or fundamental `do;/end;` blocks. In other words, RNN quickly forgets that it is inside a string literal or that it needs to close an opened `do;/end;` block. To address that issue we are going to use an RNN modification called a Long Short-Term Memory (LSTM)[6] network or its simplification a Gated Recurrent Unit (GRU)[7] network. Both networks work better in practice and deliver better results that RNNs[8]. From here we might use terms RNN/LSTM/GRU interchangeably.

[5]See Bengio u.a. (1994)
[6]See Hochreiter; Schmidhuber (1997); and Olah (2015)
[7]See Cho u.a. (2014)
[8]See Chung u.a. (2014)

DATA

To train such a network, we will need a huge corpus of SAS source code. We are going to use the following publicly available sources:

1. **GitHub**. GitHub contains a lot of SAS source code. The quality of such source code is questionable – some repositories contain a lot of very strange SAS macros, other ones contain a SAS code that is just erroneous and has syntax errors. We will talk later on how to address these issues. Besides, we developed a way of automation the process of SAS code download:
   1. Get a list of repositories with SAS code by querying GitHub Activity public dataset using Google BigQuery.
   2. Download selected repositories using a Python script.
2. **SAS Documentation** is another trusted source of SAS source code with many examples. *Hint:* XPath for crawling `//pre[@class="xis-codeFragment"]` (disclaimer: at the moment of writing the paper – 23 March 2019).
3. **SAS Technical Support FTP server** is another *hidden* resource that contains many *unusual* SAS programs from various areas – very good for training set variance.

Using the sources mentioned above may be questionable from the legal point of view because of their Terms of Use. Therefore this information is provided without warranty of any kind. The authors of this paper are not liable for any claim, damages or other liability arising from, out of or in connection with the methods of gathering data from the above sources.

Besides of the publicly available data sources any proprietary codebase can serve as a good training set – RNN will learn patterns used in that particular codebase.

Before we start training RNNs we are going to clean data in the following way:

1. Toss out all comments – we do not want an NLP problem inside an NLP problem. Although character-level RNNs can learn quite interesting patterns even from comments[9], for autocompletion functionality, this will only harm.
2. Consider only SAS macro-free code.
3. DATALINES Statements will be removed too.
4. Remove procedures with `submit;/endsubmit;` blocks.

From experience, the above clean-up techniques are the major ones that influence RNNs accuracy, especially at the early stages of training.

Besides the clean-up, we passed everything through the SASLint linter[10] in order to make sure that the SAS source code is syntactically correct and will not bias an RNN.

Additionally, we would like to mention several notes about a case-insensitivity of SAS. A variety of ways how same keywords used in SAS (like `data _null_;/DATA _null_;/Data _Null_;`) will bring much variance into our data and the character-level RNN would need to fit all these patterns. However, the token-level RNN, which will be the final aim of this paper, is case-agnostic therefore we are not going to focus on this.

CHARACTER-LEVEL RNN

The first model we are going to try is a character-level RNN. Taking into account the simplicity of this model and variety NLP tasks it is applicable for – it shows pretty good results. The vocabulary a char RNN consists of all characters encountered in the corpus. Each character

[9] See *Linux Source Code* section from Karpathy (2015)
[10] See Khorlo (2018)

is encoded as a one-hot vector. Some rare characters can be dropped and replaced by the so-called <UNK> vector. For instance, if our corpus consists only from lowercase English letters, the vocabulary will look like:

$$"a" = (1, 0, ..., 0) \in \mathbb{R}^{26}$$
$$"b" = (0, 1, ..., 0) \in \mathbb{R}^{26}$$
$$...$$
$$"z" = (0, 0, ..., 1) \in \mathbb{R}^{26}$$

Vocabulary size, in this case, will be 26. At a single timestep, the model will take an input vector (input layer), previous state (hidden layer) as input and will compute the current state and output vector which is going to be a probability distribution of the next character:

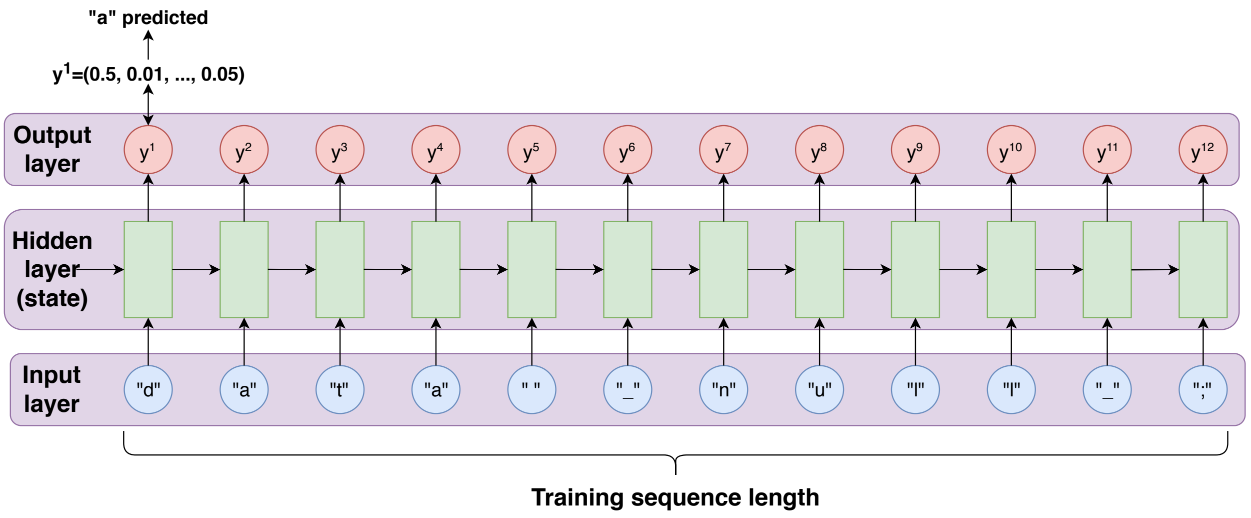

"a" predicted
y1=(0.5, 0.01, ..., 0.05)
Output layer
y1 y2 y3 y4 y5 y6 y7 y8 y9 y10 y11 y12
Hidden layer (state)
Input layer
"d" "a" "t" "a" " " "_" "n" "u" "l" "l" "_" ";"
Training sequence length

Figure 4: Forward propagation through time by RNN. Hidden layer can be a stack of several RNN layers (deep RNN). However, the last layer is always a dense layer that will give us a vector of our vocabulary dimension with logits.

At test time, we are going to feed an input (can have an arbitrary length) and get a probability distribution for a next character, from which we are going to sample a character and feed it right back into the next input. The sampling can be repeated as many times until we get the text of desirable length:

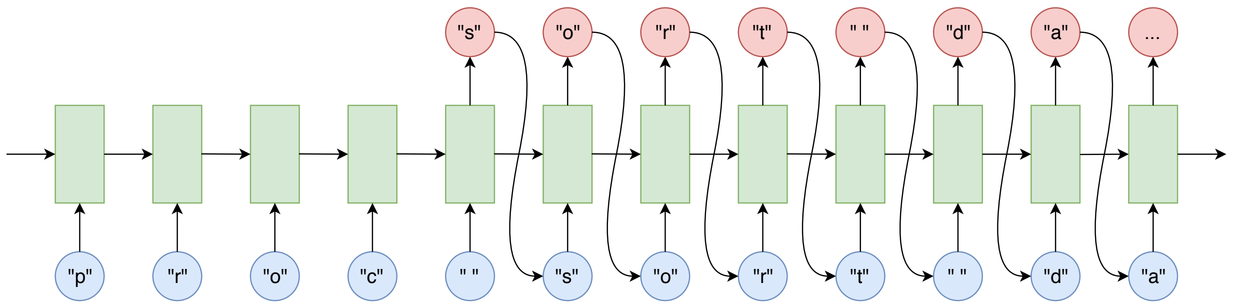

"s" "o" "r" "t" " " "d" "a" ...
"p" "r" "o" "c" " " "s" "o" "r" "t" " " "d" "a"

Figure 5: Sampling process.

We trained a 1-layer GRU with 1024 hidden units (approx. 4 million parameters) with a sequence length of 100. A cross-entropy loss, which the classic choice of a cost function for a softmax on the output layer, was optimized using Adam optimizer with mini-batch gradient descent. As a standard solution to exploding gradient problem, a gradient clipping technique was applied.

The samples are quite impressive. The following code was sampled from the trained RNN. The code is entirely fake, and no one has seen it before. The RNN knows only characters and parameters it learned from training, which in fact are just numbers. The sample is completely generated (hallucinated) by the model.

```
data medi_rdi;
  length subjid $50 fatestcd $8 invpeam1 $20;
  set freqs;
  by subjid;
  array cols{*} col1-col8 {1};
run;

options compress=binary_2;

data num;
  set means_work;
  blbmin = .;
run;

data pr0;
  length Bimabel $200;
  merge freqs new_ds;
  by tralsp_raw;
  if in1 and length_newall_db = "CEMPLUREK" and varname ^= 'CALINE BILSSTT'
    then label = 'Median';
run;

data qc_amb;
 attrib STUDYID   label="Wame of History [a]" length=8
        LBSPID    label="Specimen Type" length=$10
        MHBDSYCD  label="Low Rate of First/Location" length=$200;
  set comporiginals(in=ext);
  by usubjid;
  if first.usubjid then output;
  do j = 1 to dig(value);
    stat = catx(";", put(col1, best.), col2);
    col1 = input(put(trtpn, 5.), best.)/(-16958*25);
    dty_df = age_day;
    end;
  end;
run;
```

By the way, there is no guarantee that samples are syntactically correct and even if they are, they will likely not compile. However, it is quite remarkable that it learned so much from the source, all these indentation, function names, idiomatic structures like merge + by, attrib statement at the beginning of the DATA Steps, `first.variable` – all these things are very specific to SAS source code. What is also very important here that the model learned how to open/close quote for string literal, this knowledge we are going to use for building a token-level RNN. More samples like the one above can be found in the accomplishing GitHub repository for SAS Global Forum 2019 Proceedings[11].

TOKEN-LEVEL RNN

As we have seen in the previous section character-level RNN can quite good learn patterns from the data, which makes them especially advantageous since they work with **any** kind of input. For instance, we can use any other language, not SAS, or pass chat transcripts and try to generate a chatbot. The vocabulary size for the char-level RNN from the previous section has less than 100 "words" (single characters) which is very small for the modern NLP application measures. Another approach used in NLP applications is to treat whole words as input vectors instead of characters. SAS language has many keywords like `data`, `proc`, `print`, `set`, `if`, etc. We are going to interpret them as distinct words in the vocabulary.

Tokenization is a well-established process for structured languages parsing. We are going to use SAS Parser from SASLint project[12] to tokenize SAS source code corpus. Following is the example of a SAS program tokenization:

```
proc sort data=ae2;                        | PROC SORT DATA = ID ;
    by usubjid aestdtc aeendtc;            | BY ID ID ID ;
run;                                       | RUN ;
                                           |
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 / UNIQUE ;
  data = 123;                              | ID = <INT> ;
run;                                       | RUN ;
```

The vocabulary would be significantly bigger than in character-level RNN, but each word (token) will contain more structural information about the program. The model architecture will look like:

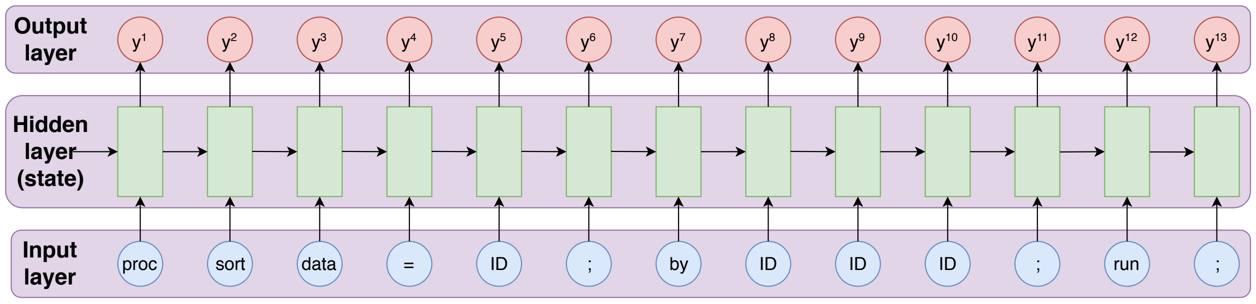


Figure 6: RNN feeds in tokens instead of characters

This approach brings a natural way of interpreting tokens in programming languages, besides it also introduces some uncertainties like identifiers, string or numerical literals. Surely, replacing all dataset/variables/formats identifiers by `ID` token would remove all referential information from the input. Therefore we not going to do this – we will distinguish between

[11] https://github.com/sascommunities/sas-global-forum-2019
[12] Khorlo (2018)

different identifiers. On the other hand, leaving a long tail of unique identifiers would increase the size of the vocabulary[13], and makes it infeasible to train RNN on such large vocabulary. There are several ways of dealing with this[14], but we are going to use knowledge from the previous section where a character-level RNN showed a good performance in learning structures like brackets, string literals (quotes). The method of representing identifier-like tokens is shown below as an example of coding a variable `foo`:

$$\texttt{<VAR>} \longrightarrow \texttt{F} \longrightarrow \texttt{O} \longrightarrow \texttt{O} \longrightarrow \texttt{B} \longrightarrow \texttt{A} \longrightarrow \texttt{R} \longrightarrow \texttt{</VAR>}$$

Where <VAR>, </VAR> are special open/closing tokens. Inside letters are represented as a sequence of single character tokens. Same we are going to apply for dataset, format, user-defined function and other identifiers. All string literals will be represented as <STR> tokens.

We trained a 2-layer GRU with 1024 hidden units (approx. 10 million parameters) with a sequence length of 200. Hyperparameters are same as for char-level RNN. Samples can be found in the accomplishing GitHub repository for SAS Global Forum 2019 Proceedings[15].

AUTOCOMPLETE

Now, our token-level RNN is trained and ready to provide autocompletion suggestions. Knowing the previous N tokens, we are going to feed them in and sample from the network. This sample is provided as the autocomplete suggestion to a user. Since we can sample sequences of arbitrary length, we are going to bound this process until a semicolon token is obtained. Suppose we have the following code – "`word = scan`" and trigger an autocompletion after "`scan`":

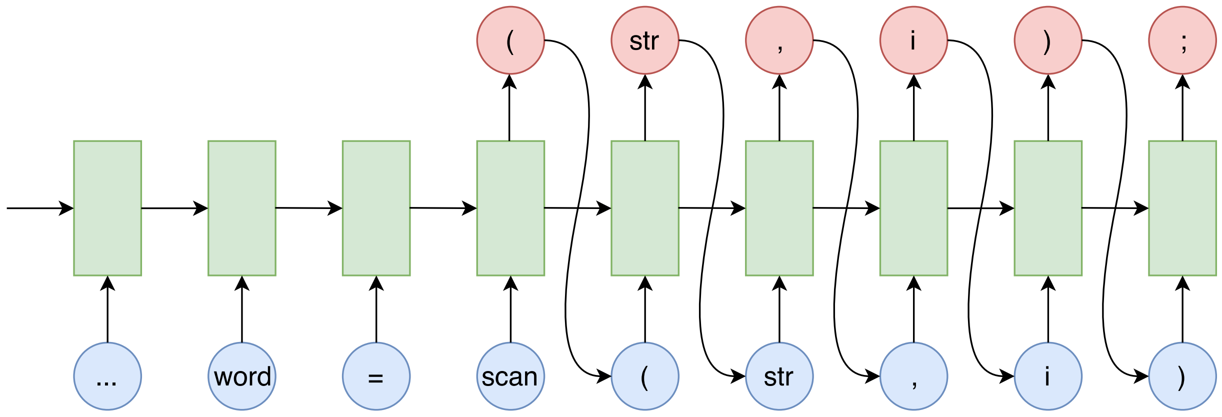


Figure 7: The process of sampling the autocompletion suggestion for "`word = scan`". Note that `word`, `str`, `i` are variables and are represented as sequences of tokens, they are convoluted for display purposes.

We can sample as many suggestions as very like. The above example selects the most *probable* sample from the RNN, although, we can sample another suggestion by selecting other tokens from the RNN output's probability distribution.

[13]Allamanis; Sutton (2013)
[14]Allamanis; Sutton (2013); Bhoopchand u.a. (2016); Karampatsis; Sutton (2019)
[15]https://github.com/sascommunities/sas-global-forum-2019

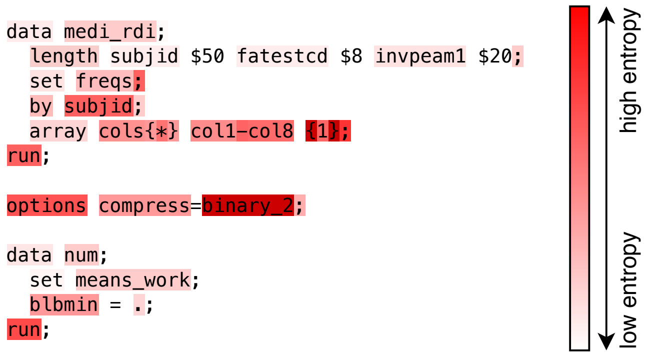
ENTROPY

We have seen how the RNN can be used to build an autocompletion engine, where the key idea was in selecting the next highest probable token from a probability distribution. What we are going to do in this chapter 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? This measure of irregularity is called 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 the current token, ln – natural logarithm.

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. 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.



```
data medi_rdi;
  length subjid $50 fatestcd $8 invpeam1 $20;
  set freqs;
  by subjid;
  array cols{*} col1-col8 {1};
run;

options compress=binary_2;

data num;
  set means_work;
  blbmin = .;
run;
```

Figure 8: Entropy heatmap for a sample generated by character-level RNN. Background colour represents the entropy of a token. High entropy (red) means suspicious, low entropy (white) means a token is regular. 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.

CONCLUSIONS & FUTURE WORK

Implementation details of the work done in this papers can are open-sourced in the accomplishing GitHub repository for SAS Global Forum 2019 Proceedings[16]. RNN was implemented using Keras[17]. The model can be trained and run on SAS® Viya® with the help of DLPy[18] Python package that can import Keras models[19].

[16]https://github.com/sascommunities/sas-global-forum-2019
[17]Keras is a high-level neural networks API – https://keras.io/
[18]DLPy - SAS Viya Deep Learning API for Python – https://github.com/sassoftware/python-dlpy
[19]SAS (2018)

Demos can be found at the following pages:

- autocompletion – https://saslint.com/rnn-autocomplete
- entropy heatmap – https://saslint.com/rnn-entropy

As we have seen RNN is a very powerful tool for modelling the source code. With such less preparation, we have been able to derive an autocompletion engine without any need to program it explicitly.

The next improvements steps and future work

- Existing autocompleting engines can be improved with this approach by providing a most probable suggestion first (recommended completion items), instead of the just an alphabetical list of keywords[20].
- Use Bidirectional Recurrent Neural Networks (BRNN). As we have seen from the heatmap, some tokens are quite unexpected for the RNN (e.g. run). However, they are pretty typical in those contexts. BRNN are a common choice in this situation and will likely solve that problem and improve the accuracy by taking into account not only backwards (as a traditional RNN), but also look forward.
- Use Abstract Syntax Trees and Graphs as source code representations. ASTs and Graphs contain more structural information about the source code thus should improve accuracy. Although, these methods are much harder to implement.

CONTACT

- Alona Bulana

  alona.bulana@gmail.com
- Igor Khorlo

  igor.khorlo@gmail.com

[20]Silver (2018); Wilson-Thomas (2018a); Wilson-Thomas (2018b)

REFERENCES

Allamanis, Miltiadis; Sutton, Charles (2013): Mining Source Code Repositories at Massive Scale Using Language Modeling. In: Reihe MSR '13, 2013207–216, URL: http://dl.acm.org/citation.cfm?id=2487085.2487127 [Accessed: 23.3.2019]

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]

Bhoopchand, Avishkar; Rocktäschel, Tim; Barr, Earl; Riedel, Sebastian (2016): Learning python code suggestion with a sparse pointer network. In: *arXiv preprint arXiv:1611.08307*, 2016, URL: https://arxiv.org/abs/1611.08307 [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]

Karampatsis, Rafael-Michael; Sutton, Charles (2019): Maybe Deep Neural Networks are the Best Choice for Modeling Source Code. In: *arXiv preprint arXiv:1903.05734*, 2019, URL: https://arxiv.org/abs/1903.05734 [Accessed: 23.3.2019]

Karpathy, Andrej (2015): The Unreasonable Effectiveness of Recurrent Neural Networks. 2015, URL: http://karpathy.github.io/2015/05/21/rnn-effectiveness/ [Accessed: 23.3.2019]

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]

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

SAS, Institute Inc. (2018): Using SAS DLPy to Define and Convert Keras Models. 2018, URL: https://github.com/sassoftware/python-dlpy/blob/master/examples/keras_model_conversion/KerasModelConversion.ipynb [Accessed: 23.3.2019]

Silver, Amanda (2018): Introducing Visual Studio IntelliCode. 2018, URL: https://devblogs.microsoft.com/visualstudio/introducing-visual-studio-intellicode/ [Accessed: 23.3.2019]

Wilson-Thomas, Mark (2018a): Visual Studio IntelliCode now infers coding conventions for consistent code. 2018, URL: https://devblogs.microsoft.com/visualstudio/visual-studio-intellicode-inferring-coding-conventions-for-consistent-code/ [Accessed: 23.3.2019]

Wilson-Thomas, Mark (2018b): Visual Studio IntelliCode supports more languages and learns from your code. 2018, URL: https://devblogs.microsoft.com/visualstudio/visual-studio-intellicode-supports-more-languages-and-learns-from-your-code/ [Accessed: 23.3.2019]