Supplementary Material for "Learning and Evaluating Contextual Embedding of Source Code"

Aditya Kanade [*1 2] **Petros Maniatis** [*2] **Gogul Balakrishnan** [2] **Kensen Shi** [2]

A. Open-Sourced Artifacts

We release data and some source-code utilities at https://github.com/google-research/google-research/tree/master/cubert. The repository contains the following:

GitHub Manifest A list of all the file versions we included into our pre-training corpus, after removing files similar to the fine-tuning corpus[1], and after deduplication. The manifest can be used to retrieve the file contents from GitHub or Google's BigQuery. This dataset was retrieved from Google's BigQuery on June 21, 2020.

Vocabulary Our subword vocabulary, computed from the pre-training corpus.

Pre-trained Models Pre-trained models on the pre-training corpus, after 1 and 2 epochs, for examples of length 512, and the BERT Large architecture.

Task Datasets Datasets containing training, validation, and testing examples for each of the 6 tasks. For the classification tasks, we provide original source code and classification labels. For the localization and repair task, we provide subtokenized code, and masks specifying the targets.

Fine-tuned Models Fine-tuned models for the 6 tasks. Fine-tuning was done on the 1-epoch pre-trained model. For each classification task, we provide the checkpoint with highest validation accuracy; for the localization and repair task, we provide the checkpoint with highest localization and repair accuracy. These are the checkpoints we used to evaluate on our test datasets, and to compute the numbers in the main paper.

Code-encoding Library We provide code for tokenizing Python code, and for producing inputs to CuBERT's pre-training and fine-tuning models.

Localization-and-repair Fine-tuning Model We provide a library for constructing the localization-and-repair model, on top of CuBERT's encoder layers. For the classification tasks, the model is identical to that of BERT's classification fine-tuning model.

Please see the `README` for details, file encoding and schema, and terms of use.

B. Data Preparation for Fine-Tuning Tasks

B.1. Label Frequencies

All four of our binary-classification fine-tuning tasks had an equal number of buggy and bug-free examples. The Exception task, which is a multi-class classification task, had a different number of examples per class (i.e., exception types). For the Exception task, we show the breakdown of example counts per label for our fine-tuning dataset splits in Table 1.

B.2. Fine-Tuning Task Datasets

In this section, we describe in detail how we produced our fine-tuning datasets (Section 3.4 of the main paper).

A common primitive in all our data generation is splitting a Python module into functions. We do this by parsing the Python file and identifying function definitions in the Abstract Syntax Tree that have no other function definition between themselves and the root of the tree. The resulting functions include functions defined at module scope, but also methods of classes and subclasses. Not included are functions defined within other function and method bodies, or methods of classes that are, themselves, defined within other function or method bodies.

We do not filter functions by length, although task-specific data generation may filter out some functions (see below). When generating examples for a fixed-length pre-training or fine-tuning model, we prune all examples to the maximum target sequence length (in this paper we consider 128, 256,

[*]Equal contribution [1]Indian Institute of Science, Bangalore, India [2]Google Brain, Mountain View, USA. Correspondence to: Aditya Kanade <kanade@iisc.ac.in>, Petros Maniatis <maniatis@google.com>.

Proceedings of the 37^{th} International Conference on Machine Learning, Online, PMLR 119, 2020.

[1]https://github.com/google-research-datasets/eth_py150_open

Exception Type	Test	Validation	Train		
			100%	66%	33%
ASSERTION_ERROR	155	29	323	189	86
ATTRIBUTE_ERROR	1,372	274	2,444	1,599	834
DOES_NOT_EXIST	7	2	3	3	2
HTTP_ERROR	55	9	104	78	38
IMPORT_ERROR	690	170	1,180	750	363
INDEX_ERROR	586	139	1,035	684	346
IO_ERROR	721	136	1,318	881	427
KEY_ERROR	1,926	362	3,384	2,272	1,112
KEYBOARD_INTERRUPT	232	58	509	336	166
NAME_ERROR	78	19	166	117	60
NOT_IMPLEMENTED_ERROR	119	24	206	127	72
OBJECT_DOES_NOT_EXIST	95	16	197	142	71
OS_ERROR	779	131	1,396	901	459
RUNTIME_ERROR	107	34	247	159	80
STOP_ITERATION	270	61	432	284	131
SYSTEM_EXIT	105	16	200	120	52
TYPE_ERROR	809	156	1,564	1,038	531
UNICODE_DECODE_ERROR	134	21	196	135	63
VALIDATION_ERROR	92	16	159	96	39
VALUE_ERROR	2,016	415	3,417	2,232	1,117

Table 1. Example counts per class for the Exception Type task, broken down into the dataset splits. We show separately the 100% train dataset, as well as its 33% and 66% subsamples used in the ablations.

512, and 1,024 subtokenized sequence lengths). Note that if a synthetically generated buggy/bug-free example pair differs only at a location beyond the target length (say on the 2,000-th subtoken), we still retain both examples. For instance, for the Variable-Misuse Localization and Repair task, we retain both buggy and bug-free examples, even if the error and/or repair locations lie beyond the end of the maximum target length. During evaluation, if the error or repair locations fall beyond the length limit of the example, we count the example as a model failure.

B.2.1. Reproducible Data Generation

We make pseudorandom choices at various stages in fine-tuning data generation. It was important to design a pseudorandomness mechanism that gave (a) reproducible data generation, (b) non-deterministic choices drawn from the uniform distribution, and (c) order independence. Order independence is important because our data generation is done in a distributed fashion (using Apache Beam), so different pseudorandom number generator state machines are used by each distributed worker.

Pseudorandomness is computed based on an experiment-wide seed, but is independent of the order in which examples are generated. Specifically, to make a pseudorandom choice about a function, we hash (using MD5) the seed and the function data (its source code and metadata about its provenance), and use the resulting hash as a uniform pseudo-random value from the function, for whatever needs the data generator has (e.g., in choosing one of multiple choices). In that way, the same function will always result in the same choices given a seed, regardless of the order in which each function is processed, thereby ensuring reproducible dataset generation.

To select among multiple choices, we hash the function's pseudorandom value along with all choices (sorted in a canonical order) and use the digest to compute an index within the list of choices. Note that given two choices over different candidates but for the same function, independent decisions will be drawn. We also use such order-independent pseudorandomness when subsampling datasets (e.g., to generate the validation datasets). In those cases, we hash a sample with the seed, as above, and turn the resulting digest into a pseudorandom number in $[0, 1]$, which can be used to decide given a target sampling rate.

B.2.2. Variable-Misuse Classification

A variable use is any mention of a variable in a load scope. This includes a variable that appears in the right-hand side of an assignment, or a field dereference. We regard as *defined* all variables mentioned either in the formal arguments of a function definition, or on the left-hand side of an assignment. We do not include in our defined variables those declared in module scope (i.e., globals).

	Commutative	Non-Commutative
Arithmetic	+, *	-, /, %
Comparison	==, !=, is, is not	<, <=, >, >=
Membership		in, not in
Boolean	and, or	

Table 2. Binary operators.

To decide whether to generate examples from a function, we parse it, and collect all variable-use locations, and all defined variables, as described above. We discard the function if it has no variable uses, or if it defines fewer than two variables; this is necessary, since if there is only one variable defined, the model has no choice to make but the default one. We also discard the function if it has more than 50 defined variables; such functions are few, and tend to be auto-generated. For any function that we do not discard, i.e., an *eligible* function, we generate a buggy and a bug-free example, as described next.

To generate a buggy example from an eligible function, we choose one variable use pseudorandomly (see above how multiple-choice decisions are done), and replace its current occupant with a different pseudorandomly-chosen variable defined in the function (with a separate multiple-choice decision).

Note that in the work by Vasic et al. (2019), a buggy and bug-free example pair was generated for *every* variable use in an eligible function. In the work by Hellendoorn et al. (2020), a buggy and bug-free example pair was generated for *up to three* variable uses in an eligible function, i.e., some functions with one use would result in one example pair, whereas functions with many variable uses would result in three example pairs. In contrast, our work produces *exactly one* example pair for every eligible function. Eligibility was defined identically in all three projects.

B.2.3. Wrong Binary Operator

This task considers both commutative and non-commutative binary operators (unlike the Swapped-Argument Classification task). See Table 2 for the full list, and note that we have excluded relatively infrequent operators, e.g., the Python integer division operator `//`.

If a function has no binary operators, it is discarded. Otherwise, it is used to generate a bug-free example, and a single buggy example as follows: one of the operators is chosen pseudorandomly (as described above), and a different operator chosen to replace it from the same row of Table 2. So, for instance, a buggy example would only swap == with `is`, but not with `not in`, which would not type-check if we performed static type inference on Python.

We take appropriate care to ensure the code parses after a bug is introduced. For instance, if we swap the operator in the expression `1==2` with `is`, we ensure that there is space between the tokens (i.e., `1 is 2` rather than the incorrect `1is2`), even though the space was not needed before.

B.2.4. Swapped Operand

Since this task targets swapping the arguments of binary operators, we only consider non-commutative operators from Table 2.

Functions without eligible operators are discarded, and the choice of the operator to mutate in a function, as well as the choice of buggy operator to use, are done as above, but limiting choices only to non-commutative operators.

To avoid complications due to format changes, we only consider expressions that fit in a single line (in contrast to the Wrong Binary Operator Classification task). We also do not consider expressions that look the same after swapping (e.g., `a - a`).

B.2.5. Function-Docstring Mismatch

In Python, a function docstring is a string literal that directly follows the function signature and precedes the main function body. Whereas in other common programming languages, the function documentation is a comment, in Python it is an actual, semantically meaningful string literal.

We discard functions that have no docstring from this dataset, or functions that have an empty docstring. We split the rest into the function definition without the docstring, and the docstring summary (i.e., the first line of text from its docstring), discarding the rest of the docstring.

We create bug-free examples by pairing a function with its own docstring summary.

To create buggy examples, we pair every function with another function's docstring summary, according to a global pseudorandom permutation of all functions: for all i, we combine the i-th function (without its docstring) with the P_i-th function's docstring summary, where P is a pseudorandom permutation, under a given seed. We discard pairings in which $i == P[i]$, but for the seeds we chose, no such pathological permuted pairings occurred.

B.2.6. Exception Type

Note that, unlike all other tasks, this task has no notion of buggy or bug-free examples.

We discard functions that do not have any `except` clauses in them.

For the rest, we collect all locations holding exception types within `except` clauses, and choose one of those locations to query the model for classification. Note that a single

`except` clause may hold a comma-separated list of exception types, and the same type may appear in multiple locations within a function. Once a location is chosen, we replace it with a special `__HOLE__` token, and create a classification example that pairs the function (with the masked exception location) with the true label (the removed exception type).

The count of examples per exception type can be found in Table 1.

B.2.7. Variable Misuse Localization and Repair

The dataset for this task is identical to that for the Variable-Misuse Classification task (Section B.2.2). However, unlike the classification task, examples contain more features relevant to localization and repair. Specifically, in addition to the token sequence describing the program, we also extract a number of boolean input masks:

- A *candidates* mask, which marks as True all tokens holding a variable, which can therefore be either the location of a bug, or the location of a repair. The first position is always a candidate, since it may be used to indicate a bug-free program.
- A *targets* mask, which marks as True all tokens holding the correct variable, for buggy examples. Note that the correct variable may appear in multiple locations in a function, therefore this mask may have multiple True positions. Bug-free examples have an all-False targets mask.
- An *error-location* mask, which marks as True the location where the bug occurs (for buggy examples) or the first location (for bug-free examples).

All the masks mark as True some of the locations that hold variables. Because many variables are subtokenized into multiple tokens, if a variable is to be marked as True in the corresponding mask, we only mark as True its first subtoken, keeping trailing subtokens as False.

C. Attention Visualizations

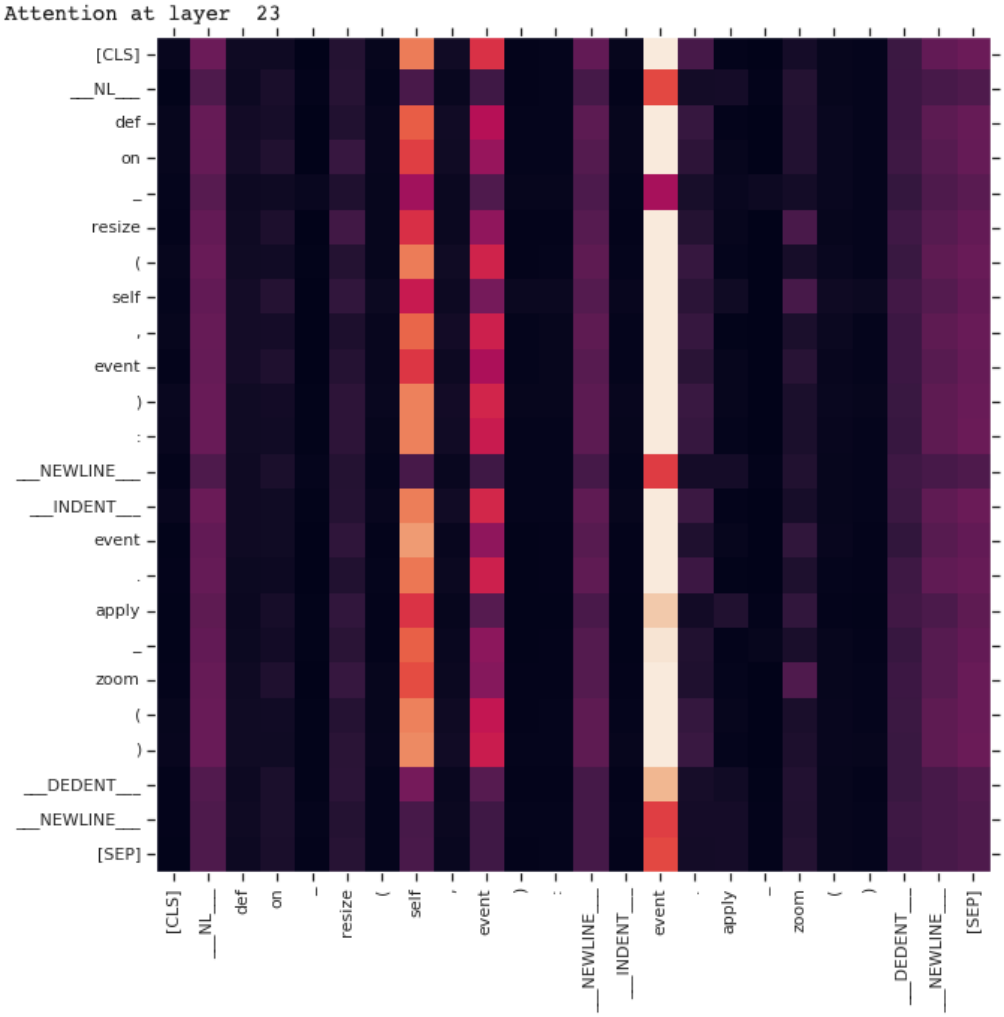
In this section, we provide sample code snippets used to test the different classification tasks. Further, Figures 1–5 show visualizations of the attention matrix of the last layer of the fine-tuned CuBERT model (Coenen et al., 2019) for the code snippets. In the visualization, the Y-axis shows the query tokens and X-axis shows the tokens being attended to. The attention weight between a pair of tokens is the maximum of the weights assigned by the multi-head attention mechanism. The color changes from dark to light as weight changes from 0 to 1.

References

Coenen, A., Reif, E., Yuan, A., Kim, B., Pearce, A., Viégas, F., and Wattenberg, M. Visualizing and Measuring the Geometry of BERT. *ArXiv*, abs/1906.02715, 2019.

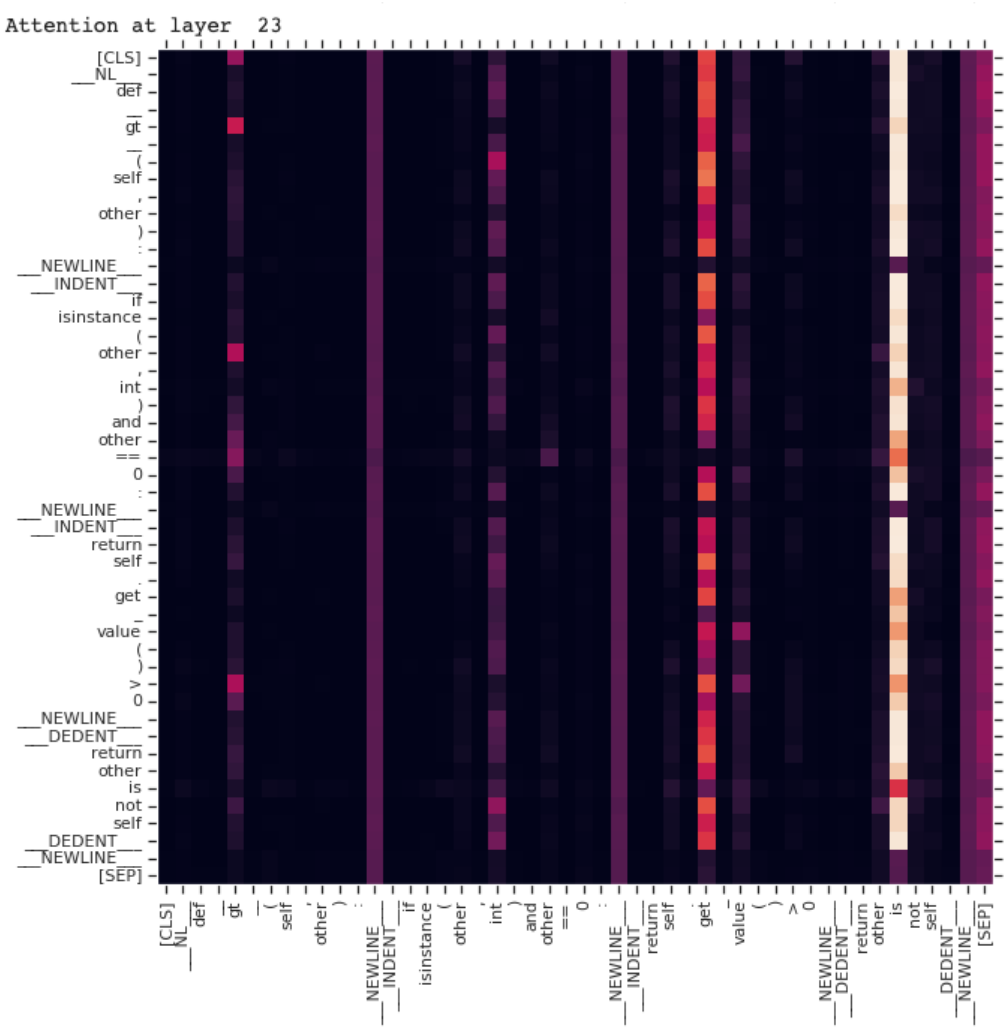
Hellendoorn, V. J., Sutton, C., Singh, R., Maniatis, P., and Bieber, D. Global relational models of source code. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=B1lnbRNtwr.

Vasic, M., Kanade, A., Maniatis, P., Bieber, D., and Singh, R. Neural program repair by jointly learning to localize and repair. *CoRR*, abs/1904.01720, 2019. URL http://arxiv.org/abs/1904.01720.




```
def on_resize(self, event):
  event.apply_zoom()
```

Figure 1. Variable Misuse Example. In the code snippet, '`event.apply_zoom`' should actually be '`self.apply_zoom`'. The CuBERT variable-misuse model correctly predicts that the code has an error. As seen from the attention map, the query tokens are attending to the second occurrence of the '`event`' token in the snippet, which corresponds to the incorrect variable usage.




```
def__gt__(self,other):
  if isinstance(other,int)and other==0:
    return self.get_value()>0
  return other is not self
```

Figure 2. Wrong Operator Example. In this code snippet, '`other is not self`' should actually be '`other < self`'. The CuBERT wrong-binary-operator model correctly predicts that the code snippet has an error. As seen from the attention map, the query tokens are all attending to the incorrect operator '`is`'.

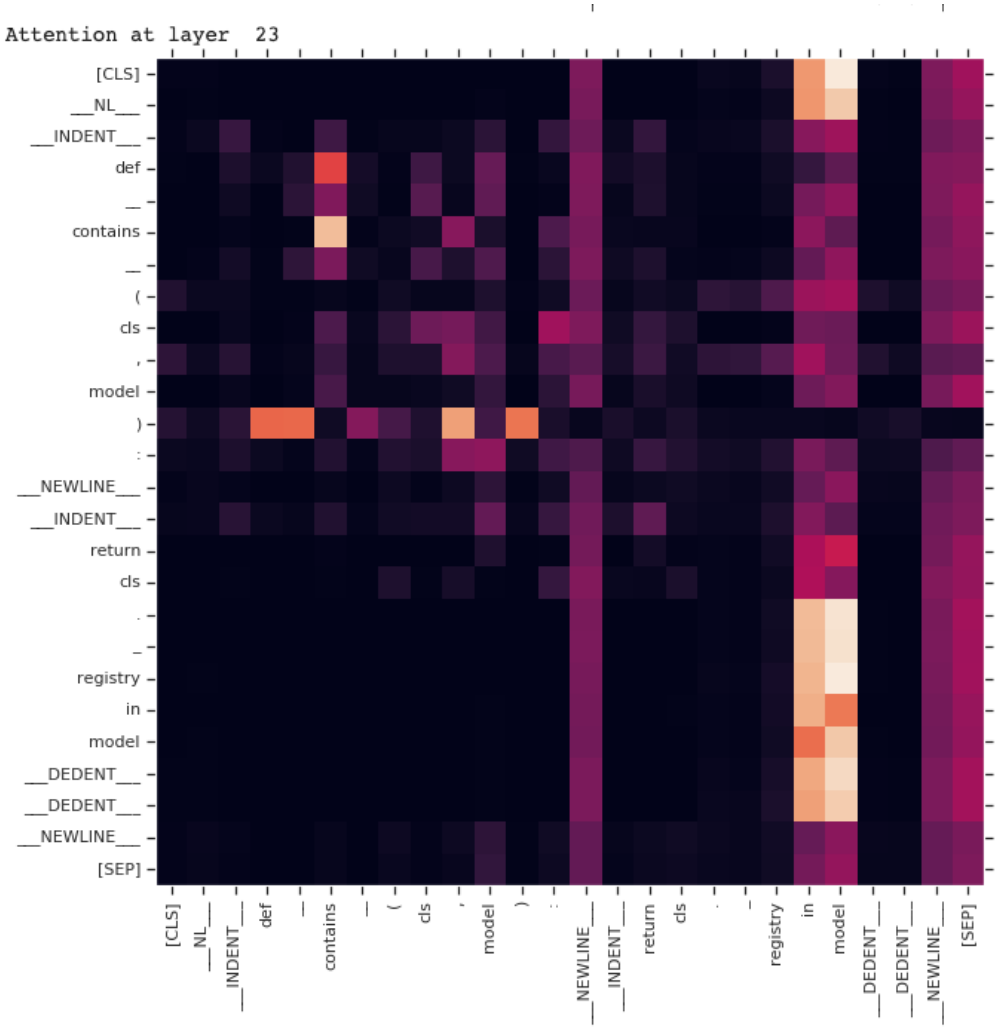
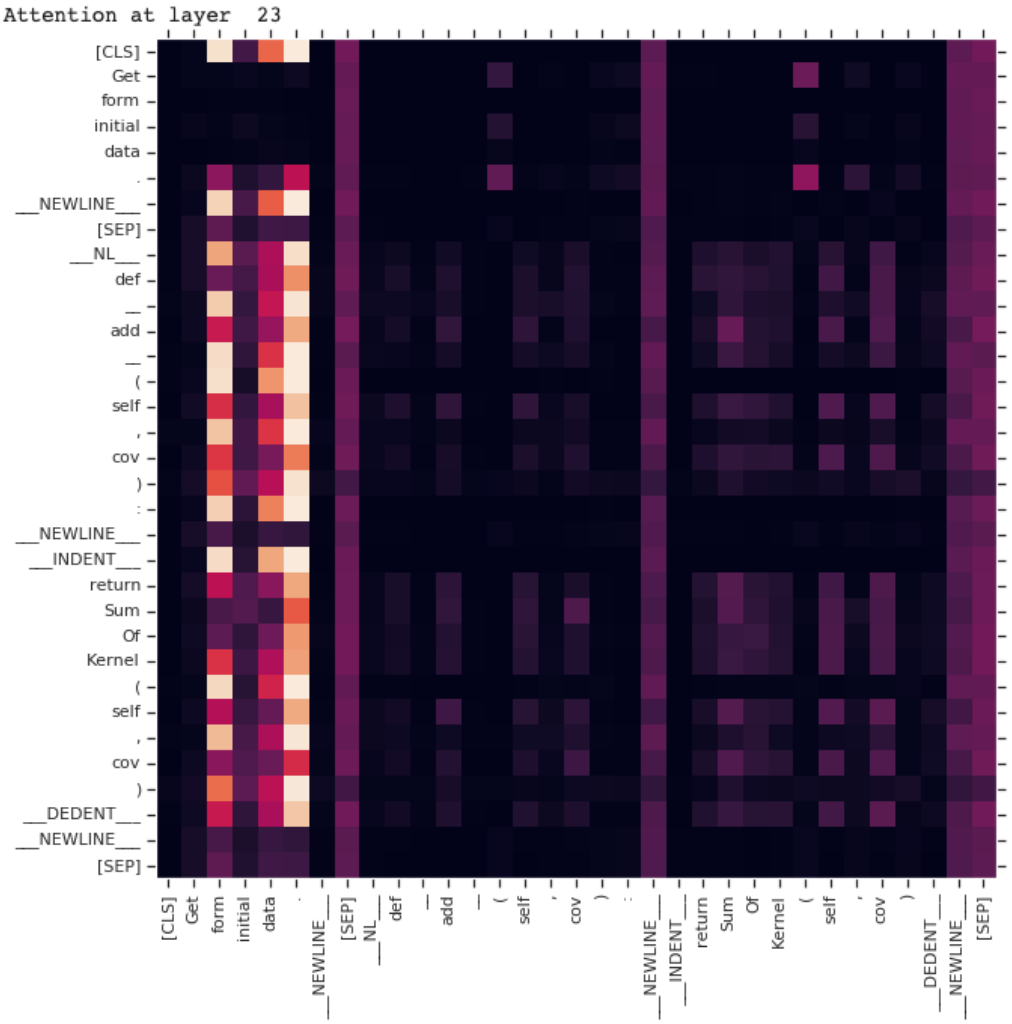
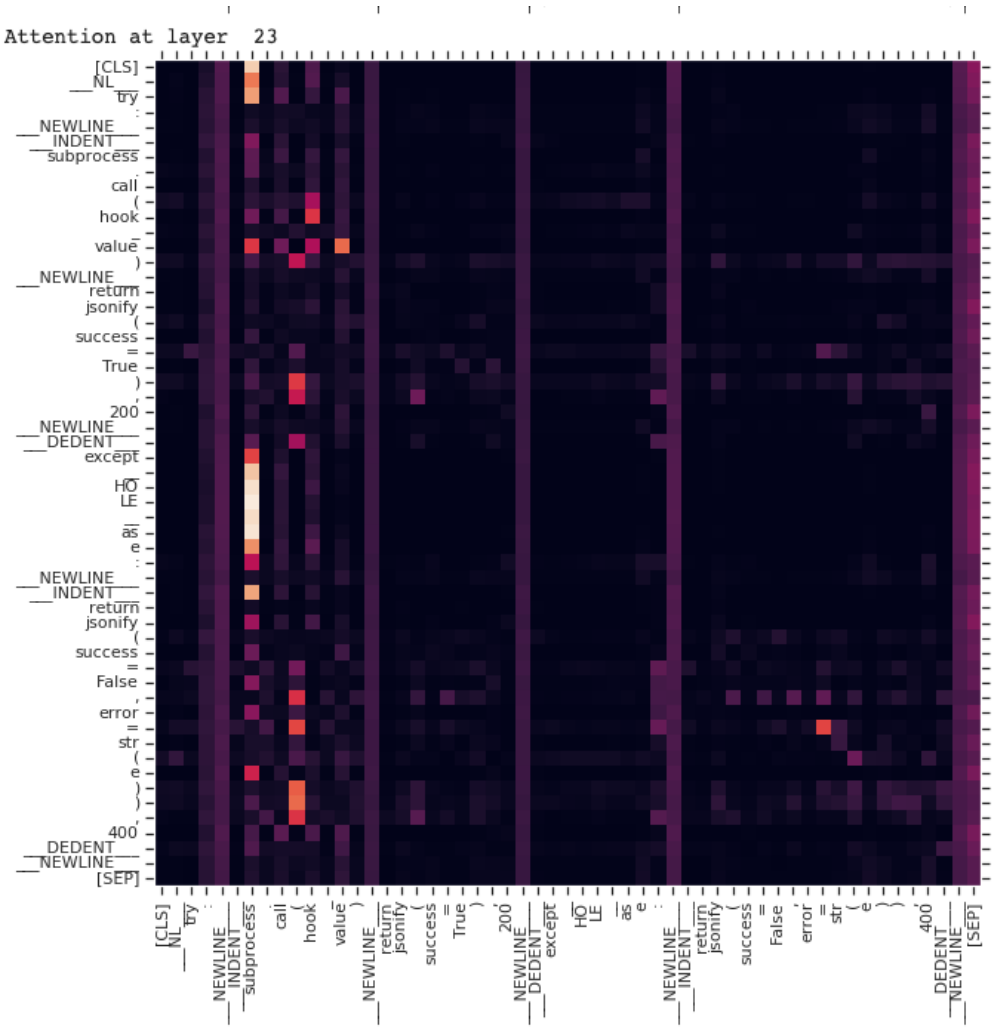


Figure 3. Swapped Operand Example. In this code snippet, the return statement should be '`model in cls._registry`'. The swapped-operand model correctly predicts that the code snippet has an error. The query tokens are paying substantial attention to '`in`' and the second occurrence of '`model`' in the snippet.




```
Docstring: 'Get form initial data.'
Function:
def__add__(self,cov):
  return SumOfKernel(self,cov)
```

Figure 4. Function Docstring Example. The CuBERT function-docstring model correctly predicts that the docstring is wrong for this code snippet. Note that most of the query tokens are attending to the tokens in the docstring.




```
try:
  subprocess.call(hook_value)
  return jsonify(success=True), 200
except __HOLE__ as e:
  return jsonify(success=False,
    error=str(e)), 400
```

Figure 5. Exception Classification Example. For this code snippet, the CuBERT exception-classification model correctly predicts '`__HOLE__`' as '`OSError`'. The model's attention matrix also shows that '`__HOLE__`' is attending to '`subprocess`', which is indicative of an OS-related error.