

Nalin: Learning from Runtime Behavior to Find Name-Value Inconsistencies in Jupyter Notebooks

Jibesh Patra
University of Stuttgart
Germany
jibesh.patra@gmail.com

Michael Pradel
University of Stuttgart
Germany
michael@binaervarianz.de

ABSTRACT

Variable names are important to understand and maintain code. If a variable name and the value stored in the variable do not match, then the program suffers from a *name-value inconsistency*, which is due to one of two situations that developers may want to fix: Either a correct value is referred to through a misleading name, which negatively affects code understandability and maintainability, or the correct name is bound to a wrong value, which may cause unexpected runtime behavior. Finding name-value inconsistencies is hard because it requires an understanding of the meaning of names and knowledge about the values assigned to a variable at runtime. This paper presents Nalin, a technique to automatically detect name-value inconsistencies. The approach combines a dynamic analysis that tracks assignments of values to names with a neural machine learning model that predicts whether a name and a value fit together. To the best of our knowledge, this is the first work to formulate the problem of finding coding issues as a classification problem over names and runtime values. We apply Nalin to 106,652 real-world Python programs, where meaningful names are particularly important due to the absence of statically declared types. Our results show that the classifier detects name-value inconsistencies with high accuracy, that the warnings reported by Nalin have a precision of 80% and a recall of 76% w.r.t. a ground truth created in a user study, and that our approach complements existing techniques for finding coding issues.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools; Software post-development issues;**

KEYWORDS

Neural software analysis, identifier names, learning-based bug detection

ACM Reference Format:

Jibesh Patra and Michael Pradel. 2021. Nalin: Learning from Runtime Behavior to Find Name-Value Inconsistencies in Jupyter Notebooks. In *ICSE '22: The 44th International Conference on Software Engineering, May 21–29, 2022, Pittsburgh, PA, USA*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA

© 2021 Association for Computing Machinery.

ACM ISBN 978-1-4503-XXXX-X/18/06... \$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

21–29, 2022, Pittsburgh, PA, USA. ACM, New York, NY, USA, 13 pages.
<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 INTRODUCTION

Variable names are a means to convey the intended semantics of code. Because meaningful names are crucial for the understandability and maintainability of code [15], developers generally try to name a variable according to the value(s) it refers to. Names are particularly relevant in dynamically typed languages, e.g., Python and JavaScript, where the lack of types forces developers to rely on names, e.g., to understand what types of values a variable stores.

Unfortunately, the name and the value of a variables sometimes do not match, which we refer to as a *name-value inconsistency*. A common reason is a *misleading name* that is bound to a correct value. Because such names make code unnecessarily hard to understand and maintain, developers may want to replace them with more meaningful names. Another possible reason is that a meaningful name refers to an *incorrect value*. Because such values may propagate through the program and cause unexpected behavior, developers should fix the corresponding code.

The following illustrates the problem with two motivating examples, both found during our evaluation on real-world Python code [49]. As an example of a misleading name consider the following code:

```
log_file = glob.glob('/var/www/some_file.csv')
```

The right-hand side of the assignment yields a list of file names, which is inconsistent with the name of the variable it gets assigned to, because `log_file` suggests a single file name. The code is even more confusing since this specific call to `glob` will return a list with at most one file name. That is, a cursory reader of the code may incorrectly assume this file name to be stored in the `log_file` variable, whereas it is actually wrapped into a list. To clarify the meaning of the variable, it could be named, e.g., `log_files` or `log_file_list`, or the developer could adapt the right-hand side of the assignment by retrieving the first (and only) element from the list. We find misleading names to be the most common reason for name-value inconsistencies.

Less common, but perhaps even worse, are name-value inconsistencies caused by an incorrect value, as in the following example:

```
train_size = 0.9 * iris.data.shape[0]
test_size = iris.data.shape[0] - train_size
train_data = data[0:train_size]
```

The code tries to divide a dataset into training and test sets. Names like `train_size` are usually bound to non-negative integer values. However, the above code assigns the value 135.0 to the `train_size` variable, i.e., a floating point value. Unfortunately, this value causes

the code to crash at the third line, where `train_size` is used as an index to slice the dataset, but indices for slicing must be integers. While the root cause and the manifestation of the crash are close to each other in this simple example, in general, incorrect values may propagate through a program and cause hard to understand misbehavior.

Finding name-value inconsistencies is difficult because it requires both understanding the meaning of names and realizing that a value that occurs at runtime does not match the usual meaning of a name. As a result, name-value inconsistencies so far are found mostly during some manual activity. For example, a developer may point out a misleading name during code review, or a developer may stumble across an incorrect value during debugging. Because developer time is precious, tool support for finding name-value inconsistencies is highly desirable.

This paper presents *Nalin*, an approach for detecting name-value inconsistencies automatically. The approach combines dynamic program analysis with deep learning. At first, a dynamic analysis keeps track of assignments during an execution and gathers pairs of names and values the names are bound to. Then, a neural model predicts whether a name and a value fit together. When the dynamic analysis observes a name-value pair that the neural model predicts to not fit together, then the approach reports a warning about a likely name-value inconsistency.

While simple at its core, realizing the *Nalin* idea involves four key challenges:

- C1 Understanding the semantics of names and how developers typically use them. The approach addresses this challenge through a learned token embedding that represents semantic similarities of names in a vector space. For example, the embedding maps the names `train_size`, `size`, and `len` to similar vectors, as they refer to similar concepts.
- C2 Understanding the meaning of values and how developers typically use them. The approach addresses this challenge by recording runtime values and by encoding them into a vector representation based on several properties of values. The properties include a string representation of the value, its type, and type-specific features, such as the shape of multi-dimensional numeric values.
- C3 Pinpointing unusual name-value pairs. We formulate this problem as a binary classification task and train a neural model that predicts whether a name and a value match. To the best of our knowledge, this work is the first to detect coding issues through neural classification over names and runtime values.
- C4 Obtaining a dataset for training an effective model. The approach addresses this challenge by considering observed name-value pairs as correct examples, and by creating incorrect examples by combining names and values through a statistical, type-guided sampling that is likely to yield an incorrect pair.

Our work relates to learning-based bug detectors [6, 18, 33, 47, 57], which share the idea to classify code as correct or incorrect. However, we are the first to focus on name-value inconsistencies, whereas prior work targets other kinds of problems. *Nalin* also relates to learned models that predict missing identifier names [12, 17, 48]. Our work differs by analyzing code with names supposed to be meaningful, instead of targeting obfuscated or compiled code. Finally, there are static analysis-based approaches for

finding inconsistent method names [26, 34, 38] and other naming issues [22]. A key difference to all the above work is that *Nalin* is based on dynamic instead of static analysis, allowing it to learn from runtime values, which static analysis can only approximate. One of the few existing approaches that learn from runtime behavior [56] aims at finding vector representations for larger pieces of code, but cannot pinpoint name-value inconsistencies.

We train *Nalin* on 780k name-value pairs and evaluate it on 10k previously unseen pairs from real-world Python code extracted from Jupyter notebooks. The model effectively distinguishes consistent from inconsistent examples, with an F1 score of 0.89. Comparing the classifications by *Nalin* to a ground truth gathered in a study with eleven developers shows that the reported inconsistencies have a precision of 80% and a recall of 76%. Most of the inconsistencies detected in real-world code are due to misleading names, but there also are some inconsistencies caused by incorrect values. Finally, we show that the approach complements state-of-the-art static analysis-based tools that warn about frequently made mistakes, type-related issues, and name-related bugs.

In summary, this paper contributes the following:

- An automatic technique to detect name-value inconsistencies.
- The first approach to find coding issues through neural machine learning on names and runtime behavior.
- A type-guided generation of negative examples that improves upon a purely random approach.
- Empirical evidence that the approach effectively identifies name-value pairs that developers perceive as detrimental to the understandability and maintainability of the code.

2 OVERVIEW

This section describes the problem we address and gives an overview of our approach. *Nalin* reasons about *name-value pairs*, i.e., pairs of a variable name and a value that gets assigned to the variable. The problem we address is to identify name-value pairs where the name is not a good fit for the value, which we call *inconsistent name-value pairs*. Identifying such pairs is an inherently fuzzy problem: Whether a name fits a value depends on the conventions that programmers follow when naming variables. The fuzziness of the problem motivates a data-driven approach [45], where we use the vast amounts of available programs as guidance for what name-value pairs are common and what name-value pairs stand out as inconsistent.

Broadly speaking, *Nalin* consists of six components and two phases, illustrated in Figure 1. During the training phase, the approach learns from a corpus of executable programs a neural classification model, which then serves during the prediction phase for identifying name-value inconsistencies in previously unseen programs. The following illustrates the six components of the approach with some examples. A detailed description follows in Section 3.

Given a corpus of executable programs, the first component is a dynamic analysis of assignments of values to variables. For each assignment during the execution of the program, the analysis extracts the variable name, the value assigned to the variable, and several properties of the value, e.g., the type, length, and shape. As illustrated in Figure 1, properties that do not exist for a particular value are represented by *null*. For example, the analysis extracts

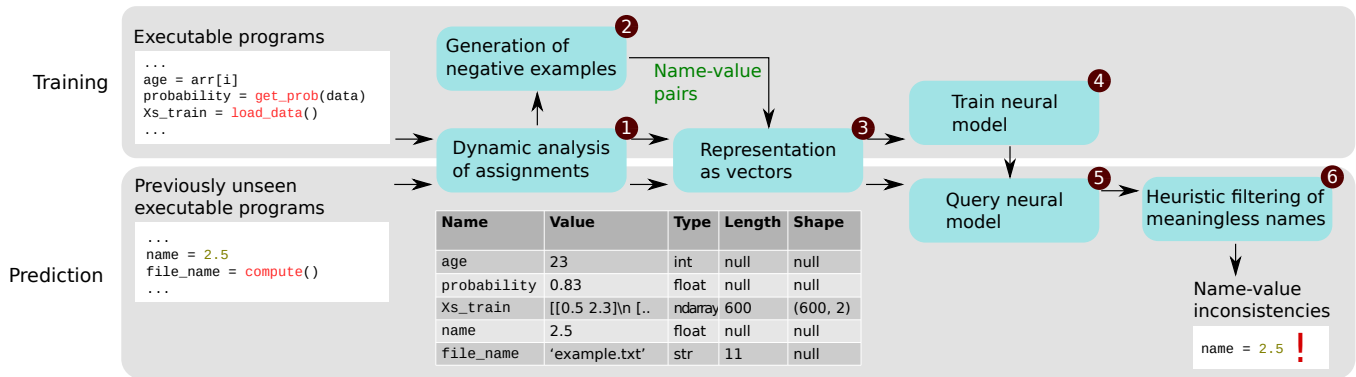


Figure 1: Overview of the approach.

the length of the assigned value for `Xs_train`, but not for `age` and `probability`, as the corresponding values are primitives that do not have a length.

While the name-value pairs obtained by the dynamic analysis serve as positive examples, the second component generates negative examples that combine names and values in an unusual and likely inconsistent way. The motivation behind generating negative examples is that Nalin trains a classification model in a supervised manner, i.e., the approach requires examples of both consistent and inconsistent name-value pairs. Using the example pairs in Figure 1, one negative example would be the name `Xs_train` paired with the floating point value 0.83, which indeed is an unusual name-value pair. Our approach for generating negative examples is a probabilistic algorithm that biases the selection of unusual values toward unusual types based on the types of values that are usually observed with a name. The first and second component together address challenge C4 from the introduction, i.e., obtaining a dataset for training an effective model.

The third component of Nalin addresses challenges C1 and C2, i.e., “understanding” the semantics of names and values. To this end, the approach represents names and values as vectors that preserve their meaning. To represent identifier names, we build on learned token embeddings [13], which map each name into a vector while preserving the semantic similarities of names [54]. For example, the vector of `probability` will be close to the vectors of names `probab` and `likelihood`, because these names refer to similar concepts. To represent values, we present a neural encoding of values based on their string representation, type, and other properties.

Given the vector representations of name-value pairs, the fourth component trains a neural model to distinguish positive from negative examples. The result is a classifier that, once trained with sufficiently many examples, addresses challenge C3. The fifth component of the approach queries the classifier with vector representations of name-value pairs extracted from previously unseen programs, producing a set of pairs predicted to be inconsistent. The final component heuristically filters pairs that are likely false positives, and then reports the remaining pairs as warnings to the developer. For the two new assignments shown in Figure 1, the trained classifier will correctly identify the assignment `name = 2.5` as unusual and raises a warning.

3 APPROACH

The following presents the components of Nalin outlined in the previous section in more detail.

3.1 Dynamic Analysis of Assignments

The goal of this component is to gather name-value pairs from a corpus of programs. Our analysis focuses on assignments because they associate a value with the name of a variable. One option would be to statically analyze all assignments in a program. However, a static analysis could capture only those values where the right-hand side of an assignment is a literal, but would miss many other assignments, e.g., when the right-hand side is a complex expression or function call. In the code corpus used in our evaluation, we find that 90% of all assignments have a value other than a primitive literal on the right-hand side, i.e., a static analysis could not gather name-value pairs from them. Instead, Nalin uses a dynamic analysis that observes all assignments during the execution of a program. Besides the benefit of capturing assignments that are hard to reason about statically, a dynamic analysis can easily extract additional properties of values, such as the length or shape, which we find to be useful for training an effective model.

3.1.1 Instrumentation and Data Gathering. To dynamically analyze assignments, Nalin instruments and then executes the programs in the corpus. For instrumentation, the analysis traverses the abstract syntax tree of a program and augments all assignments to a variable with a call to a function that records the name of the variable and the assigned value.

As runtime values can be arbitrarily complex, the analysis can extract only limited information about a value. We extract four properties of each value, which we found to be useful for training an effective model, but extending the approach to gather additional properties of values is straightforward. Slightly abusing the term “pair” to include the properties extracted for each value, the analysis extracts the following information:

Definition 1 (Name-value pair). A name-value pair is a tuple (n, v, τ, l, s) , where n denotes the variable name on the left hand side, v is a string representation of the value, τ represents the type of the value, and l and s represent the length and shape of the value, respectively.

The string representation builds upon Python’s built-in string conversion, which often yields a meaningful representation because developers commonly use this representation, e.g., for debugging. The type of values is relevant because it allows Nalin to find type-related mistakes, which otherwise remain easily unnoticed in a dynamically typed language. Length here refers to the number of items present in a collection or sequence type value, which is useful, e.g., to enable the model to distinguish empty from non-empty collections. Since some common data types are multidimensional the *shape* refers to the number of items present in each dimension. The table in Figure 1 shows examples of name-value pairs gathered by the analysis. We show in the evaluation how much the extracted properties contribute to the overall effectiveness of the model.

3.1.2 Filtering and Processing of Name-Value Pairs.

Merge Types. We observe that the gathered data forms a long-tailed distribution of types. One of the reasons is the presence of many similar types, such as Python’s dictionary type *dict* and its subclass *defaultdict*. To help the model generalize across similar types, we reduce the overall number of types by merging some of the less frequent types. To this end, we first choose the ten most frequent types present in the dataset. For the remaining types, we replace any types that are in a subclass relationship with one of the frequent types by the frequent type. For example, consider a name-value pair (*stopwords*, *frozenset*({"all", "afterwards", "eleven", ...})), *frozenset*, 337, null). Because type *frozenset* is not among the ten most frequent types, but type *set* is, we change the name-value pair into (*stopwords*, *frozenset*({"all", "afterwards", "eleven", ...})), *set*, 337, null).

Filter Meaningless Names. An underlying assumption of Nalin is that developers use meaningful variable names. Unfortunately, some names are rather cryptic, such as variables called *a* or *ts_pd*. Such names help neither our model nor developers in deciding whether a name fits the value it refers to, and hence, we filter likely meaningless names. The first type of filtering considers the length of the variable names and discards any name-value pairs where the name is less than three characters long. The second type of filtering is similar to the first one, except that it targets names composed of multiple subtokens, such as *ts_pd*. We split names at underscores¹, and remove any name-value pairs where each subtoken has less than three characters.

3.2 Generation of Negative Examples

The gathered name-value pairs provide numerous examples of names and values that developers typically combine. Nalin uses supervised learning to train a classification model that distinguishes consistent, or positive, name-value pairs from inconsistent, or negative, pairs. Based on the common assumption that most code is correct, we consider the name-value pairs extracted from executions as positive examples. The following presents two techniques for generating negative examples. First, we explain a purely random technique, followed by a type-guided technique that we find to yield a more effective training dataset.

Algorithm 1 Create a negative example

Input: Name-value pair (n, v, τ, l, s) , dataset D of all pairs

Output: Negative example (n, v', τ', l', s')

```

1:  $F_{global} \leftarrow$  Compute from  $D$  a map from types to their frequency
2:  $F_{name} \leftarrow$  Compute from  $D$  and  $n$  a map from types observed
   with  $n$  to their frequency
3:  $T_{name} \leftarrow \emptyset$  ▷ Types seen with  $n$ 
4:  $T_{name\_infreq} \leftarrow \emptyset$  ▷ Types infrequently seen with  $n$ 
5: for each  $(\bar{\tau} \mapsto f) \in F_{name}$  do
6:    $T_{name} \leftarrow \bar{\tau}$ 
7:   if  $f \leq$  threshold then
8:      $T_{name\_infreq} \leftarrow \bar{\tau}$ 
9:  $T_{all} \leftarrow dom(F_{global})$  ▷ All types ever seen
10:  $T_{cand} = (T_{all} \setminus T_{name}) \cup T_{name\_infreq}$  ▷ Types never or
   infrequently seen with  $n$ 
11:  $\tau' \leftarrow weightedRandomChoice(T_{cand}, F_{global})$ 
12:  $v', l', s' \leftarrow randomChoice(D, \tau')$ 
13: return  $(n, v', \tau', l', s')$ 

```

3.2.1 Purely Random Generation. Our purely random algorithm for generating negative examples is straightforward. For each name-value pair (n, v, τ, l, s) , the algorithm randomly selects another name-value pair (n', v', τ', l', s') from the dataset. Then, the algorithm creates a new negative example by combining the name of the original pair and the value of the randomly selected pair, which yields (n, v', τ', l', s') .

While simple, the purely random generation of negative examples suffers from the problem of creating many name-value pairs that do fit well together. The underlying root cause is that the distribution of values and types is long-tailed, i.e., the dataset contains many examples of similar values among the most common types. For example, consider a name-value pair gathered from an assignment `num = 23`. When creating a negative example, the purely random algorithm may choose a value gathered from another assignment `age = 3`. As both values are positive integers, they both fit the name `num`, i.e., the supposedly negative example actually is a legitimate name-value pair. Having many such legitimate, negative examples in the training data makes it difficult for a classifier to discriminate between consistent and inconsistent name-value pairs.

3.2.2 Type-Guided Generation. To mitigate the problem of legitimate, negative examples that the purely random generation algorithm suffers from, we present a type-guided algorithm for creating negative examples. The basic idea is to first select a type that a name is infrequently observed with, and to then select a random value among those observed with the selected type. Algorithm 1 shows the type-guided technique for creating a negative example for a given name-value pair. The inputs to the algorithm are a name-value pair (n, v, τ, l, s) and the complete dataset D of positive name-value pairs.

The first two lines of Algorithm 1 create two helper maps, which map types to their frequency. The F_{global} map assigns each type to its frequency across the entire dataset D , whereas the F_{name} map assigns each type to how often it occurs with the name n of the positive example. Next, lines 3 to 8 populate two sets of types. The

¹<https://www.python.org/dev/peps/pep-0008/#function-and-variable-names>

- ◇ Given name-value pair:
`years = [2011, 2012, 2013, 2014]`
 $(n, v, \tau, l, s) = (\text{years}, [2011, 2012, 2013, 2014], \text{list}, 4, \text{null})$
- ◇ All types years has been in the dataset and their frequencies:
 $F_{\text{name}} = \{ \text{list}: 235, \text{ndarray}: 59, \text{int}: 33, \text{float}: 7, \text{dict}: 5, \text{tuple}: 4, \text{set}: 1 \}$
- ◇ Infrequent types for years:
 $T_{\text{name_infreq}} = \{ \text{float}, \text{dict}, \text{tuple}, \text{set} \}$
- ◇ Global frequencies of types infrequently or never seen with years:
 $F_{\text{global}} = \{ \text{str}: 89337, \text{bool}: 5385, \text{float}: 71244, \text{dict}: 21654, \dots \}$
- ◇ Weighted random selection of a target type:
 $\tau' = \text{float}$
- ◇ Random selection of a float value from the dataset:
 $(n, v', \tau', l', s') = (\text{years}, 1.8, \text{float}, \text{null}, \text{null})$

Figure 2: Steps for creating a negative example.

first set, T_{name} , is populated with all types ever observed with name n . The second set, $T_{\text{name_infreq}}$, is populated with all types that are infrequently observed with name n . “Infrequent” here means that the frequency of the type among all name-value tuples with name n is below some threshold, which is 3% in the evaluation. The goal of selecting types that are infrequent for a particular name is to create negative examples that are unusual, and hence, likely to be inconsistent.

The remainder of the algorithm (lines 9 to 13) picks a type to be used for the negative example and then creates a negative name-value pair by combining the name n with a value of that type. To this end, the algorithm computes all candidate types, T_{cand} , that are either never observed with name n or among the types $T_{\text{name_infreq}}$ that infrequently occur with n . The algorithm then randomly selects among the candidate types, using the global type frequency as weights for the random selection. The rationale is to choose a type that is unlikely for the name n , while following the overall distribution of types. The latter is necessary to prevent the model from simply learning to spot unlikely types, but to instead learn to find unlikely combinations of names and values. Once the target type τ' for the negative example is selected, the algorithm randomly picks a value among all values (line 12) observed with type τ' , and eventually returns a negative example that combines name n with the selected value.

Figure 2 illustrates the algorithm with an example from our evaluation. The goal is to create a negative example for a name-value pair where the name is `years`. In the dataset of positive examples, the name `years` occurs with values of types `list`, `ndarray`, `int`, etc., with the frequencies shown in the figure. For example, `years` occurs 235 times with a `list` value, but only seven times with a `float` value. Among all types that occur in the dataset, many never occur together with the name `years`, e.g., `str` and `bool`. Based on the global frequencies of types that `years` never or only infrequently occurs with, the algorithm picks `float` as the target type. Finally, a corresponding `float` value is selected from the dataset, which is `1.8` for the example, and the negative example shown at the bottom of the figure is returned.

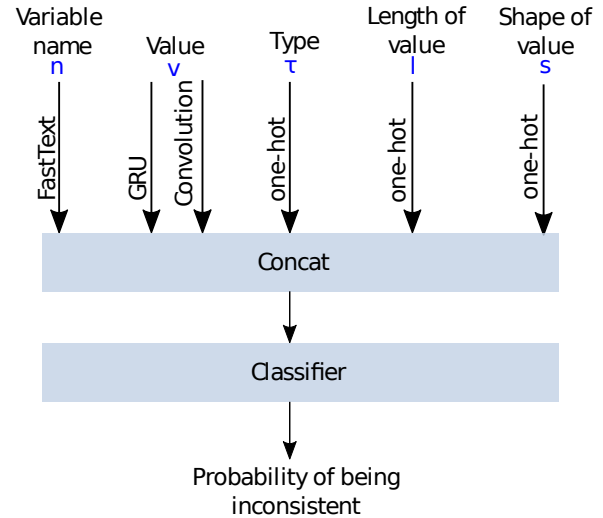


Figure 3: Architecture of the neural model.

By default, Nalin uses the type-guided generation of negative examples, and our evaluation compares it with the purely random technique. The generated negative examples are combined with the positive examples in the dataset, and the joint dataset serves as training data for the neural classifier. Due to the automated generation, a generated negative examples may coincidentally be identical to an existing positive example. In practice, the dataset used during the evaluation contains 38 instances of identical positive and negative examples out of 490,332 negative examples.

3.3 Representation as Vectors

Given a dataset of name-value pairs, each labeled either as a positive or a negative example, Nalin trains a neural classification model to distinguish the two kinds of examples. A crucial step is to represent the information in a name-value pair as vectors, which we explain in the following. The approach first represents each of the five components (n, v, τ, l, s) of a name-value pair as a vector, and then feeds the concatenation of these vectors into the classifier. Figure 3 shows an overview of the neural architecture. The following describes the vector representation in more detail, followed by a description of the classifier in Section 3.4.

Representing Variable Names. To enable Nalin to reason about the meaning of variable names, it maps each name into a vector representation that encodes the semantics of the name. For example, the representation should map the names `list_of_numbers` and `integers` to similar vectors, as both represent similar concepts, but the vector representations of the names `age` and `file_name` should differ from the previous vectors. To this end, our approach builds on pre-trained word embeddings, i.e., a learned function that maps each name into a vector. Originally proposed in natural language processing as a means to represent words [13, 36], word embeddings are becoming increasingly popular also on source code [9, 35, 39, 41, 47], where they represent individual tokens, e.g., variable names.

We build upon FastText [13], a neural word embedding known to represent the semantics of identifiers more accurately than other popular embeddings [54]. An additional key benefit of FastText is to avoid the out-of-vocabulary problem that other embeddings, e.g., Word2vec [36] suffer from, by splitting each token into n-grams and by computing a separate vector representation for each n-gram. To obtain meaningful embeddings for the Python domain, we pre-train a FastText model on token sequences extracted from the corpus Python programs used in our evaluation. Formally, the trained FastText model M , assigns to each name n a real-valued vector $M(n) \in \mathbb{R}^d$, where $d = 100$ in our evaluation.

Representing Values. The key challenge for representing the string representations of values as vectors is that there is a wide range of different values, including sequential structures, e.g., in values of types *string*, *ndarray*, *list*, and values without an obvious sequential structure, e.g., primitives and custom objects. The string representations of values may capture many interesting properties, including and beyond the information conveyed by the type of a value. For example, the string representation of an *int* implicitly encodes whether the value is a positive or negative number. Our goal when representing values as vector is to pick up such intricacies, without manually defining type-specific vector encoders.

To this end, Nalin represents value as a combination of two vector representations, each computed by a neural model that we jointly learn along with the overall classification model. On the one hand, we use a recurrent neural network (RNN) suitable for capturing sequential structures. Specifically, we apply gated recurrent units (GRU) over the sequence of characters, where each character is used as an input at every timestep. The vector obtained from the hidden state of the last timestep then serves as the representation of the complete sequence. On the other hand, we use a convolutional neural network (CNN) suitable for capturing non-sequential information about the value. Specifically, the approach applies a one-dimensional CNN over the sequence of characters, where the number of channels for the CNN is equal to the number of characters in the string representation of the value, the number of output channels is set to 100, Relu is the activation function, and a one-dimensional MaxPool layer serves as the final layer. Finally, Nalin concatenates the vectors obtained from the RNN and the CNN into the overall vector representation of the value.

Representing Types. To represent the type of a value as a vector, the approach computes a one-hot vector for each type. Each vector has a dimension equal to the number of types present in the dataset. A type is represented by setting an element to one while keeping the remaining elements set to zero. For example, if we have only three types namely *int*, *float*, and *list* in our dataset then using one-hot encoding, each of them can be represented as $[1, 0, 0]$, $[0, 1, 0]$ and $[0, 0, 1]$ respectively. For the evaluation, we set the maximum number of types to ten. More sophisticated representations of types, e.g., learned jointly with the overall model [5], could be integrated into Nalin as part of future work.

Representing Length and Shape. Length and shape are similar concepts, and hence, we represent them in a similar fashion. Because the length of a value is theoretically unbounded, we consider ten ranges of lengths and represent each of them with a one-hot vector.

Specifically, Nalin considers ranges of length 100, starting from 0 until 1,000. That is, any length between 0 and 100 will be represented by the same one-hot vector, and likewise any length greater than 1,000 will be represented by the another vector. The shape of a value is a tuple of discrete numbers, which we represent similarly to the length, except that we first multiply the elements of the shape tuple. For example, for a value of shape x, y, z , we encode $x \cdot y \cdot z$ using the same approach as for the length. For values that do not have a length or shape, we use a special one-hot vector.

3.4 Training and Prediction

Once Nalin has obtained a vector representation for each component of a name-value pair, the individual vectors are concatenated into the combined representation of the pair. We then feed this combined representation into a neural classifier that predicts the probability p of the name-value pair to be inconsistent. The classification model consists of two linear layers with a sigmoid activation function at the end. We also add a dropout with probability of 0.5 before each linear layer. We train the model with a batch size of 128, using the Adam [28] optimizer, for 15 epochs, after which the validation accuracy saturates. During training, the model is trained toward predicting $p = 0.0$ for all positive examples and $p = 1.0$ for all negative examples. Once trained, we interpret the predicted probability p as the confidence Nalin has in flagging a name-value pair as inconsistent, and the approach reports to the user only pairs with p above some threshold (Section 4.2).

3.5 Heuristic Filtering of Likely False Positives

Before reporting name-value pairs that the model predicts as inconsistent to the user, Nalin applies two simple heuristics to prune likely false positives. The heuristics aim at removing generic and meaningless names that have passed the filtering described in Section 3.1.2, such as `data` and `val_0`. The rationale is that judging whether those names match a specific value is difficult, but the goal of Nalin is to identify name-value pairs that clearly mismatch. The first heuristic removes pairs with names that contain one of the following terms, which are often found in generic names: `data`, `value`, `result`, `temp`, `tmp`, `str`, and `sample`. The second heuristic removes pairs with short and cryptic names. To this end, we tokenize names at underscores and then remove pairs with names where at least one subtoken has less than three characters.

4 EVALUATION

Our evaluation focuses on the following research questions:

- RQ1: How effective is the neural model of Nalin in detecting name-value inconsistencies?
- RQ2: Are the inconsistencies that Nalin reports perceived as hard to understand by software developers?
- RQ3: What kinds of inconsistencies does the approach find in real-world code?
- RQ4: How does our approach compare to popular static code analysis tools?
- RQ5: How does Nalin compare to simpler variants of the approach?

4.1 Experimental Setup

We implement our approach for Python as it is one of the most popular dynamically typed programming languages[1]. All experiments are run on a machine with Intel Xeon E5-2650 CPU having 48 cores, 64GB of memory and an NVIDIA Tesla P100 GPU. The machine runs Ubuntu 18.04, and we use Python 3.8 for the implementation.

The evaluation requires a large-scale, diverse, and realistic dataset of closed (i.e., include all inputs) programs. We choose one million computational notebooks in an existing dataset of Jupyter notebooks scrapped from GitHub [49]. The dataset is (i) large-scale because there are many notebooks available, (ii) diverse because they are written by various developers and cover various application domains, (iii) realistic because Jupyter notebooks are one of the most popular ways of written Python code these days, and (iv) closed because notebooks do not rely on user input. Another option would be to apply Nalin to executions of test suites, which often focus on unusual inputs though and, by definition, exercise well-tested and hence likely correct behavior.

Excluding some malformed notebooks, we convert 985,865 notebooks into Python scripts using *nbconvert*. Some of these notebooks contain only text and no code, while for others, the code has syntax errors, or the code is very short and does not perform any assignments. All of this decreases the number of Python files that Nalin can instrument, and we finally obtain 598,321 instrumented files. The instrumentation takes approximately two hours.

When gathering name-value pairs, we face general challenges related to reproducing Jupyter notebooks [55]. First, even with the installation of the 100 most popular Python packages, unresolved dependencies result in crashes during some executions. Second, some Python scripts read inputs from files, e.g., a dataset for training a machine learning model, which may not be locally available. Considering all notebooks that we can successfully execute despite these obstacles, Nalin gathers a total of 947,702 name-value pairs, of which 500,332 remain after the filtering described in Section 3.1.2. The extracted pairs come from 106,652 Python files with a total of 7,231,218 lines of non-comment, non-blank Python code. Running the instrumented files to extract name-value pairs takes approximately 48 hours.

Before running any experiments with the model, we sample 10,000 name-value pairs as a held-out *test dataset*. Unless mentioned otherwise, all reported results are on this test dataset. On the remaining 490,332 name-value pairs, we perform an 80-20 split into *training* and *validation* data. For each name-value pair present in the training, validation, and test datasets, we create a corresponding negative example, which takes two hours in total. The total number of data points used to train the Nalin model hence is about 780k. Training takes an average of 190 seconds per epoch and once trained, prediction on the entire test dataset takes about 15 seconds.

We find the name-value pairs to consist of a diverse set of values and types. There are 99.8k unique names, i.e., each name appears, on average, about 10 times. The top-5 frequent types are *list*, *ndarray*, *str*, *int*, *float*. The presence of a large number of collection types, such as *list* and *ndarray*, which usually are not fully initialized as literals shows that extracting values at run-time is worthwhile.

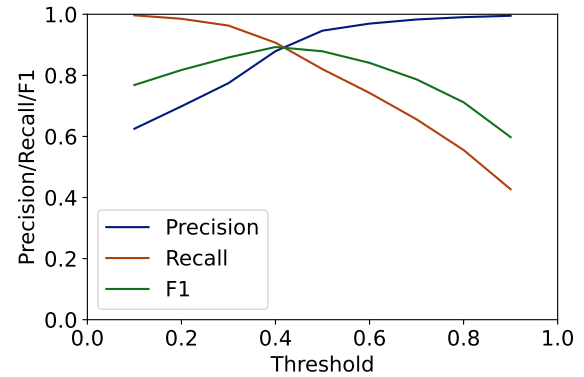


Figure 4: Precision, recall, and F1 score with different thresholds for reporting warnings.

4.2 RQ1: Effectiveness of the Trained Model

We measure the effectiveness of Nalin’s model by applying the trained model to the held-out test dataset. The output of the model can be interpreted as a confidence score that indicates how likely the model believes a given name-value pair to be inconsistent. We consider all name-value pairs $P_{warning}$ with a score above some threshold as a warning, and then measure precision and recall of the model w.r.t. the inconsistency labels in the dataset ($P_{inconsistent}$ are pairs labeled as inconsistent):

$$precision = \frac{|P_{warning} \cap P_{inconsistent}|}{|P_{warning}|}$$

$$recall = \frac{|P_{warning} \cap P_{inconsistent}|}{|P_{inconsistent}|}$$

We also compute the F1 score, which is the harmonic mean of precision and recall.

Figure 4 shows the results for different thresholds for reporting a prediction as a warning. The results illustrate the usual precision-recall tradeoff, where a user can reduce the risk of false positive warnings at the cost of finding fewer inconsistencies. The model achieves the highest F1 score of 89% at a threshold of 0.4, with a precision of 88% and a recall of 91%. Unless otherwise mentioned, we use a threshold of 0.5 as the default, which gives 87% F1 score. Out of 8,858 files in the held-out test set, 336 (3.8%) have at least one warning reported by Nalin.

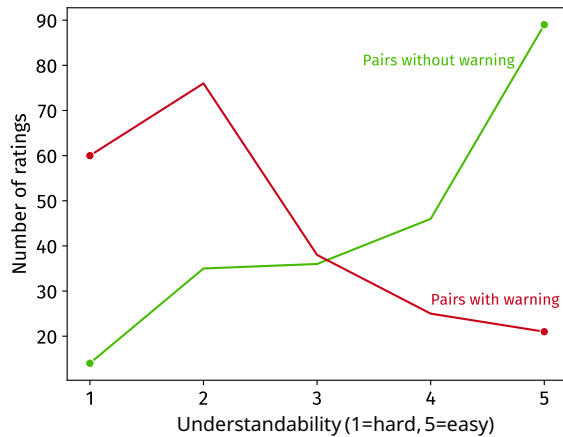
Finding 1: The model effectively identifies inconsistent name-value pairs, with a maximum F1 score of 89%.

4.3 RQ2: Study with Developers

To answer the question how well Nalin’s warnings match name-value pairs that developers perceive as hard to understand, we perform a study with eleven software developers. The participants are four PhD students and seven master-level students, all of which regularly develop software, and none of which overlaps with the authors of this paper. During the study, each participant is shown 40 name-value pairs and asked to assess each pair regarding its understandability. The participants provide their assessment on a five-point Likert scale ranging from “hard” (1) to “easy” (5), where

Developer assessment	Nalin’s prediction	
	Consistent ($P_{noWarning}$)	Inconsistent ($P_{warning}$)
Easy to understand (P_{easy})	15	4
Hard to understand (P_{hard})	5	16

(a) Nalin predictions of inconsistencies vs. developer-perceived understandability.



(b) Understandability ratings for name-value pairs with and without warnings by Nalin.

Figure 5: Results from user study.

“hard” means that the name and the value are inconsistent, making it hard to understand and maintain the code. The 40 name-value pairs consist of 20 pairs that are randomly selected from all warnings Nalin reports as inconsistent with a confidence above 80% and of 20 randomly selected pairs that the approach does not warn about. For each pair, the participants are shown the name of the variable, the value that Nalin deems inconsistent with this name, and the type of the value. In total, the study hence involves 440 developer ratings. Because what is a meaningful variable names is, to some extent, subjective, we expect some variance in the ratings provided by the participants. To quantify this variance, we compute the inter-rater agreement using Krippendorff’s alpha, which yields an agreement of 56%. That is, developers agree with a medium to high degree on whether a name-value pair is easy to understand.

Before providing quantitative results, we discuss a few representative examples. Among the name-value pairs without a warning is a variable called `DATA_URL` that stores a string containing a URL. This pair is consistently rated as easy to understand, with a mean ranking of 5.0. Among the pairs that Nalin reports as inconsistent are a variable `password_text` storing an integer value `0`, which most participants consider as hard to understand (mean rating: 1.54). Another pair that the approach warns about is a variable called `path` that stores an empty list. The study participants are rather undecided about this example, with a mean rating of 2.72.

The main question of the user study is to what extent Nalin pinpoints name-value pairs that developers also consider to be hard to understand. We address this question in two ways, first by

computing precision and recall of Nalin w.r.t. the developer ratings, and then by comparing the ratings for warnings and non-warnings.

Precision and Recall w.r.t. Developer Ratings. We assign each of the 40 name-value pairs into two sets: On the one hand, a pair is in P_{hard} if the mean rating assigned by the developers is less than three and in P_{easy} otherwise. On the other hand, a pair is in $P_{warning}$ if Nalin flags it as an inconsistency and in $P_{noWarning}$ otherwise. Table 5a shows the intersections between these sets. For example, we see that 16 of the pairs that Nalin warns about, but only 5 of the pairs without a warning, are considered to be hard to understand. We compute precision and recall as follows:

$$precision = \frac{|P_{warning} \cap P_{hard}|}{|P_{warning}|} = \frac{16}{20} = 80\%$$

$$recall = \frac{|P_{warning} \cap P_{hard}|}{|P_{hard}|} = \frac{16}{21} = 76\%$$

Ratings for Warnings vs. Non-Warnings. In addition to the pair-based metrics above, we also globally compare the ratings for pairs with and without warnings. The goal is to understand whether Nalin is effective at distinguishing between name-value pairs that developers perceive as easy and hard to understand. To this end, consider two sets of ratings: ratings $R_{warning}$ for name-value pairs that Nalin reports as inconsistent, and ratings $R_{noWarning}$ for other name-value pairs. Figure 5b compares the two sets of ratings with each other, showing how many ratings there are for each point on the 5-point Likert scale. The results show a clear difference between the two sets: “easy” is the most common rating in $R_{noWarning}$, whereas the majority of ratings in $R_{warning}$ is either “relatively hard” or “hard”. We also statistically compare $R_{warning}$ and $R_{noWarning}$ using a Mann-Whitney U-test, which shows the two sets of rankings to be extremely likely to be sampled from different populations (with a p-value of less than 0.1%).

Finding 2: Developers mostly agree with the (in)consistency predictions by Nalin. In particular, they assess 80% of the name-value pairs that the approach warns about as hard to maintain and understand.

4.4 RQ3: Kinds of Inconsistencies in Real-World Code

To better understand the kinds of name-value inconsistencies detected in real-world code, we inspect name-value pairs in the test datasets that appear as such in the code, but that are classified as inconsistent by the model. When using Nalin to search for previously unknown issues, these name-value pairs will be reported as warnings. We inspect the top-30 predictions, sorted by the probability score provided by the model, and classify each warning into one of three categories:

- *Misleading name.* Name-value pairs where the name clearly fails to match the value it refers to. These cases do not lead to wrong program behavior, but should be fixed to increase the readability and maintainability of the code.
- *Incorrect value.* Name-value pairs where the mismatch between a name and a value is due to an incorrect value being assigned. These cases cause unexpected program behavior, e.g., a program crash or incorrect output.

Table 1: Examples of warnings produced by Nalin.

Code Example	Category	Run-time value	Comment
<pre>name = 'Philip K. Dick' ... name = 2.5 if type(name) == str: print('yes')</pre>	Misleading name	2.5	A variable called name is typically holding a string, but here stores a float value.
<pre>file = os.path.exists('reference.csv') if file == False: print('Warning: ...')</pre>	Misleading name	False	The name file suggests that the variable stores either a file handle or a file name, but it here stores a boolean.
<pre>def Custom(information): prob = get_betraying_probability(information) if(prob > 1 / 2): return D elif(prob == 1 / 2): return choice([D, C]) else: return C</pre>	Incorrect value	"Corporate"	Assigning a string to a variable called prob is unusual, because prob usually refers to a probability. The value is incorrect and leads to a crash in the next line because comparing a string and a float causes a type error.
<pre>dwarF = '/Users/iaiyork/Downloads/dwar_2013_2015.txt' dwar = pd.read_csv(dwarF, sep=' ', header=None)</pre>	False positive	"/Users/.."	The value is a string that describes file path, which fits the name, where the F supposedly means "file". The model reports this false positive because it fails to understand the abbreviation.

- *False positive.* Name-value pairs that are consistent with each other, and which ideally would not be reported as a warning.

The inspection shows that 21 of the warnings correspond to misleading names, 2 are incorrect values, and 7 are false positives. That is, the majority of the reported inconsistencies are due to the name, whereas only a few are caused by an incorrect value being assigned to a meaningful name. This result is expected because incorrect behavior is easier to detect, e.g. via testing, than misleading names, for which currently few tools exist. The fact that 23 out of 30 warnings (77%) are true positives is also consistent with the developer study in RQ2.

Table 1 shows representative examples of warnings produced by Nalin. The first two examples show misleading names. For example, it is highly unusual to assign a number to a variable called name or to assign boolean to a variable called file. To the best of our knowledge, these misleading names do not cause unexpected behavior, but developers may still want to fix them to increase the readability and maintainability of the code. In the third example, Nalin produces a warning about the assignment on line 2. The value assigned during the execution is a string 'Cooperate'. Due to the string assignment, the code on line 3 crashes since the operator > does not support a comparison between a string and float. Nalin is correct in predicting this warning because the variable name prob is typically used to refer to a probability, not to a string like 'Cooperate'. The final example is a false positive, which illustrates one of the most common causes of false positives seen during our inspection, namely short (and somewhat cryptic) names for which the model fails to understand the meaning.

Finding 3: The majority of inconsistencies detected in real-world code are due to the name in a name-value pair being misleading, and occasionally also due to incorrect values.

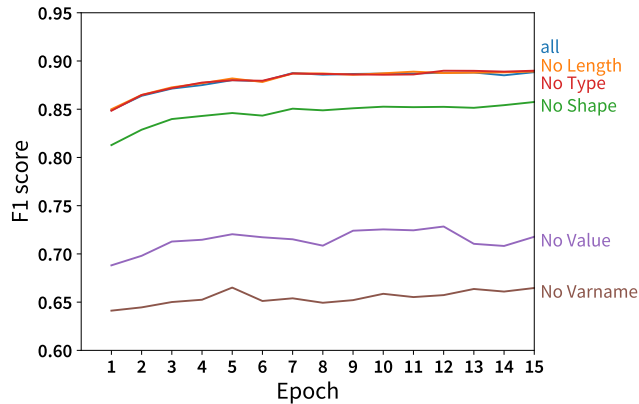
4.5 RQ4: Comparison with Previous Bug Detection Approaches

We compare Nalin to three state-of-the-art static analysis tools aimed at finding bugs and other kinds of noteworthy issues: (i) *pyre*, a static type checker for Python that infers types and uses available type annotations. We compare with *pyre* because many of the inconsistencies that Nalin reports are type-related, and hence, might also be spotted by a type checker. (ii) *flake8*, a Python linter that warns about commonly made mistakes. We compare with *flake8* because it is widely used and because linters share the goal of improving the quality of code. (iii) *DeepBugs* [47], a learning-based bug detection technique. We compare with *DeepBugs* because it also aims to find name-related bugs using machine learning, but using static instead of dynamic analysis. We run *pyre* and *flake8* using their default configurations. For *DeepBugs*, we install the "DeepBugs for Python" plugin from the marketplace of the PyCharm IDE. We apply each of the three approaches to the 30 files where Nalin has produced a warning and which have been manually inspected (RQ3). Namer [22], a recent technique for finding name-related coding issues through a combination of static analysis, pattern mining, and supervised learning would be another candidate for comparing with, but neither the implementation nor the experimental results are publicly available.

Table 2 shows the number of warnings reported by the existing tools and how many of these warnings overlap with those reported

Table 2: Comparison with existing static bug detectors.

Approach	Warnings	Warnings common with Nalin
pyre	54	1/30
flake8	1,247	0/30
DeepBugs	151	0/30

**Figure 6: Result of ablation study.**

by Nalin. We find that except one warning reported by pyre, none matches with the 30 manually inspected warnings from Nalin. The matching warning is a misleading name, shown on the first row of Table 1. The pyre type checker reports this as an “Incompatible variable type” because in the same file, the variable name is first assigned a string ‘Philip K. Dick’ and later assigned a float value 2.5. The 1,247 warnings produced by flake8 are mostly about coding style, e.g., “missing white space” and “whitespace after ‘(’”. The warnings reported by DeepBugs include possibly wrong operator usages and incorrectly ordered function arguments, but none matches the warnings reported by Nalin.

Finding 4: Nalin is complementary to both traditional static analysis-based tools and to a state-of-the-art learning-based bug detector aimed at name-related bugs.

4.6 RQ5: Comparison with Variants of the Approach

4.6.1 Type-Guided vs. Purely Random Negative Examples. The following compares the two algorithms for generating negative examples described in Section 3.2. Following the setup from RQ1, we find that the purely random generation reduces both precision and recall, leading to a maximum F1 score of 0.82, compared to 0.89 with the type-guided approach. Manually inspecting the top-30 reported warnings as in RQ2, we find 21 false positives, nine misleading names, and zero incorrect values, which clearly reduces the precision compared to the type-guided generation approach. These results confirm motivation for the type-guided algorithm (Section 3.2.2) and show that it outperforms a simpler baseline.

4.6.2 Ablation Study. We perform an ablation study to measure the importance of the different components of a name-value pair fed into the model. To this end, we set the vector representation of individual components to zero during training and prediction, and then measure the effect on the F1 score of the model. Figure 6 shows the results, where the vertical axis shows the F1 score obtained on the validation dataset at each epoch during training. Each line in Figure 6 shows the F1 score obtained while training the model keeping that particular feature set to zero. For example, the green line (“No Shape”) is for a model that does not use the shape of a value, and the blue line (“all”) is for a model that uses all components of a name-value pair. We find that the most important inputs to the model are the variable name and the string representation of the value. Removing the length or the type of a value does not significantly decrease the model’s effectiveness. The reason is that these properties can often be inferred from other inputs given to the model, e.g., by deriving the type from the string representation of a value. We confirm this explanation by removing both the type and the string representation of a value, which yields an F1 score similar to the model trained by removing only values.

Finding 5: Each component of the approach contributes to the overall effectiveness, but there is some redundancy in the properties of values given to the model.

5 THREATS TO VALIDITY

Internal Validity. Several factors may influence our results. First, the filtering of name-value pairs based on the length of names may accidentally remove short but meaningful names, such as abbreviations that are common in a specific domain. Preliminary experiments without such filtering resulted in many false positives, and we prefer false negatives over false positives to increase developer acceptance. Second, our manual classification into different kinds of inconsistencies is subject to our limited knowledge of the analyzed Python files. To mitigate this threat, the classification is done by two of the authors, discussing any unclear cases until reaching consensus.

External Validity. Several factors may influence the generalizability of our results. First, our approach is designed with a dynamically typed programming language in mind, because meaningful identifier names are particularly important in such languages. This focus and the setup of our experiments implies that we cannot draw conclusions beyond Python or beyond the kind of Python code found in Jupyter notebooks. Second, our developer study is limited to eleven participants, and other developers may assess the understandability of the name-value pairs differently. We mitigate this threat by getting eleven opinions about each name-value pair and by statistically analyzing the relevance of the results.

6 RELATED WORK

Detecting Poor Names. The importance of meaningful names during programming has been studied and established [15, 32]. There are several techniques for finding poorly named program elements, e.g., based on pre-defined rules [2], by comparing method names

against method bodies [26], and through a type inference-like analysis of names and their occurrences [30]. To improve identifier names, rule-based expansion [31], n-gram models of code [3], and learning-based techniques that compare method bodies and method names have been proposed [34, 38]. Namer [22] combines static analysis, pattern mining, and supervised learning to find name-related coding issues. Many of the above approaches focus on method names, whereas we target variables. Moreover, none of the existing approaches exploits dynamically observed values.

Predicting Names. When names are completely missing, e.g., in minified, compiled, or obfuscated code, learned models can predict them [12, 29, 48, 53]. Another line of work predicts method names given the body of a method [4, 7, 9], which beyond being potentially useful for developers serves as a pseudo-task to force a model to summarize code in a semantics-preserving way. Nalin differs by considering values observed at runtime, and not only static source code, and by checking names for inconsistencies with the values they refer to, instead of predicting names from scratch.

Name-based Program Analysis. DeepBugs introduced learning-based and name-based bug detection [47], which differs from Nalin by being purely static and by focusing on different kinds of errors. The perhaps most popular kind of name-based analysis is probabilistic type inference [59], often using deep neural network models [5, 23, 35, 46, 58] that reason about the to-be-typed code. RefiNum uses names to identify conceptual types, which further refine the usual programming language types [16]. SemSeed exploits semantic relations between names to inject realistic bugs [42]. All of the above work is based on the observation that the implicit information embedded in identifiers is useful for program analyses. Our work is the first to exploit this observation to find name-value inconsistencies.

Natural Language vs. Code. Beyond natural language in the form of identifiers, comments and documentation associated with code are another valuable source of information. iComment [51] and tComment [52] use this information to detect inconsistencies between comments and code. Our work differs by focusing on variable names instead of comments, by comparing the natural language artifact against runtime values instead of static code, and by using a learning-based approach. Another line of work uses natural language documentation to infer specifications of code [19, 37, 40], which is complementary to our work.

Learning on Code. In addition to the work discussed above, machine learning on code is receiving significant interest recently [45]. Embeddings of code are one important topic, e.g., using AST paths [10], control flow graphs [57], ASTs [60], or a combination of token sequences and a graph representation of code [24]. Our encoder of variable names could benefit from being combined with an encoding of the code surrounding the assignment using those ideas. Other work models code changes and then makes predictions about them [14, 25], or trains models for program repair [18, 21], code completion [8, 11, 27], and code search [20, 50].

Learning from Executions. Despite the recent surge of work on learning on code, learning on data gathered during executions is a relatively unexplored area. One model embeds student programs

based on dynamically observed input-output relations [43]. Wang et al.’s “blended” code embedding learning [56] combines runtime traces, which include values of multiple variables, and static code elements to learn a distributed vector representation of code. Beyond code embedding, BlankIt [44] uses a decision tree model trained on runtime data to predict the library functions that a code location may use. In contrast to these papers, our work addresses a different problem and feeds one value at a time into the model.

7 CONCLUSION

Using meaningful identifier names is important for code understandability and maintainability. This paper presents Nalin, which addresses the problem of finding inconsistencies that arise due to the use of a misleading name or due to assigning an incorrect value. The key novelty of Nalin is to learn from names and their values assigned at runtime. To reason about the meaning of names and values, the approach embeds them into vector representations that assign similar vectors to similar names and values. Our evaluation with about 500k name-value pairs gathered from real-world Python programs shows that the model is highly accurate, leading to warnings reported with a precision of 80% and recall of 76%.

Our implementation and experimental results are available: <https://github.com/sola-st/Nalin>

ACKNOWLEDGMENTS

This work was supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and Perf4JS projects.

REFERENCES

- [1] [n.d.]. TIOBE Index. <https://www.tiobe.com/tiobe-index>. Accessed: 2021-08-31.
- [2] Surafel Lemma Abebe, Sonia Haiduc, Paolo Tonella, and Andrian Marcus. 2009. Lexicon bad smells in software. In *2009 16th Working Conference on Reverse Engineering*. IEEE, 95–99.
- [3] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 281–293.
- [4] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 38–49.
- [5] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. In *PLDI*.
- [6] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. <https://openreview.net/forum?id=BJOFETxR->
- [7] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *ICML*. 2091–2100.
- [8] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. <https://openreview.net/forum?id=H1gKY09tX>
- [9] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A General Path-Based Representation for Predicting Program Properties. In *PLDI*.
- [10] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29. <https://doi.org/10.1145/3290353>
- [11] Gareth Ari Aye and Gail E. Kaiser. 2020. Sequence Model Design for Code Completion in the Modern IDE. *CoRR abs/2004.05249* (2020). arXiv:2004.05249 <https://arxiv.org/abs/2004.05249>

- [12] Rohan Bavishi, Michael Pradel, and Koushik Sen. 2018. Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts. *CoRR arXiv:1809.05193* (2018).
- [13] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. *TACL* 5 (2017), 135–146. <https://transacl.org/ojs/index.php/tacl/article/view/999>
- [14] Shaked Brody, Uri Alon, and Eran Yahav. 2020. A Structural Model for Contextual Code Changes. In *OOPSLA*.
- [15] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010. Exploring the Influence of Identifier Names on Code Quality: An Empirical Study. In *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 156–165.
- [16] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2018. RefiNym: Using Names to Refine Types. In *ESEC/FSE*.
- [17] Yaniv David, Uri Alon, and Eran Yahav. 2020. Neural reverse engineering of stripped binaries. In *OOPSLA*.
- [18] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *ICLR*.
- [19] Alberto Goffi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 213–224.
- [20] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep Code Search. In *ICSE*.
- [21] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *AAAI*.
- [22] Jingxuan He, Cheng-Chun Lee, Veselin Raychev, and Martin T. Vechev. 2021. Learning to find naming issues with big code and small supervision. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 296–311. <https://doi.org/10.1145/3453483.3454045>
- [23] V. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis. 2018. Deep Learning Type Inference. In *FSE*.
- [24] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global Relational Models of Source Code. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. <https://openreview.net/forum?id=B1lnbRNTwr>
- [25] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: Distributed Representations of Code Changes. In *ICSE*.
- [26] Einar W. Høst and Bjarte M. Østvold. 2009. Debugging Method Names. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 294–317.
- [27] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2020. Code Prediction by Feeding Trees to Transformers. *arXiv preprint arXiv:2003.13848* (2020).
- [28] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
- [29] Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A Neural Approach to Decompiled Identifier Naming. In *ASE*.
- [30] Julia L. Lawall and David Lo. 2010. An automated approach for finding variable-constant pairing bugs. In *International Conference on Automated Software Engineering (ASE)*. ACM, 103–112.
- [31] Dawn Lawrie and Dave Binkley. 2011. Expanding identifiers to normalize source code vocabulary. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 113–122.
- [32] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a Name? A Study of Identifiers. In *International Conference on Program Comprehension (ICPC)*. IEEE, 3–12.
- [33] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. 2019. Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks. In *OOPSLA*.
- [34] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Tae-young Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 1–12. <https://dl.acm.org/citation.cfm?id=3339507>
- [35] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 304–315. <https://doi.org/10.1109/ICSE.2019.00045>
- [36] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*. 3111–3119.
- [37] Manish Motwani and Yuriy Brun. 2019. Automatically generating precise Oracles from structured natural language specifications. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 188–199. <https://doi.org/10.1109/ICSE.2019.00035>
- [38] Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. 2020. Suggesting Natural Method Names to Check Name Consistencies. In *ICSE*.
- [39] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. 2017. Exploring API embedding for API usages and applications. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 438–449.
- [40] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring method specifications from natural language API descriptions. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 815–825.
- [41] Md. Rizwan Parvez, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2018. Building Language Models for Text with Named Entities. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*. 2373–2383. <https://doi.org/10.18653/v1/P18-1221>
- [42] Jibesh Patra and Michael Pradel. 2021. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 906–918. <https://doi.org/10.1145/3468264.3468623>
- [43] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas J. Guibas. 2015. Learning Program Embeddings to Propagate Feedback on Student Code. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. 1093–1102. <http://proceedings.mlr.press/v37/piech15.html>
- [44] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. 2020. BlankIt library debloating: getting what you want instead of cutting what you don't. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 164–180. <https://doi.org/10.1145/3385412.3386017>
- [45] Michael Pradel and Satish Chandra. 2022. Neural Software Analysis. *Commun. ACM* (2022). <https://arxiv.org/abs/2011.07986> To appear.
- [46] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. Type-Writer: Neural Type Prediction with Search-based Validation. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. 209–220. <https://doi.org/10.1145/3368089.3409715>
- [47] Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. *PACMPL* 2, OOPSLA (2018), 147:1–147:25. <https://doi.org/10.1145/3276517>
- [48] Veselin Raychev, Martin T. Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Principles of Programming Languages (POPL)*. 111–124.
- [49] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (Montreal QC, Canada) (CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. <https://doi.org/10.1145/3173574.3173606>
- [50] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM, 31–41.
- [51] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: Bugs or bad comments??. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 145–158.
- [52] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. 2012. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 260–269.
- [53] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar T. Devanbu. 2017. Recovering clear, natural identifiers from obfuscated JS names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 683–693.
- [54] Yaza Wainakh, Moiz Rauf, and Michael Pradel. 2021. IdBench: Evaluating Semantic Representations of Identifier Names in Source Code. In *IEEE/ACM International Conference on Software Engineering (ICSE)*.
- [55] Jiawei Wang, KUO Tzu-Yang, Li Li, and Andreas Zeller. 2020. Assessing and Restoring Reproducibility of Jupyter Notebooks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 138–149.
- [56] Ke Wang and Zhendong Su. 2020. Blended, precise semantic program embeddings. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 121–134. <https://doi.org/10.1145/3385412.3385999>
- [57] Yu Wang, Fengjuan Gao, Linzhang Wang, and Ke Wang. 2020. Learning Semantic Program Embeddings with Graph Interval Neural Network. In *OOPSLA*.

- [58] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. <https://openreview.net/forum?id=Hkx6hANTwH>
- [59] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python probabilistic type inference with natural language support. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 607–618. <https://doi.org/10.1145/2950290.2950343>
- [60] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation based on Abstract Syntax Tree. In *ICSE*.