Technical Report: Towards a Universal Code Formatter through Machine Learning

Terence Parr
University of San Francisco
parrt@cs.usfca.edu

Jurgen Vinju
Centrum Wiskunde & Informatica
Jurgen.Vinju@cwi.nl

Abstract

There are many declarative frameworks that allow us to implement code formatters relatively easily for any specific language, but constructing them is cumbersome. The first problem is that "everybody" wants to format their code differently, leading to either many formatter variants or a ridiculous number of configuration options. Second, the size of each implementation scales with a language's grammar size, leading to hundreds of rules.

In this paper, we solve the formatter construction problem using a novel approach, one that automatically derives formatters for any given language without intervention from a language expert. We introduce a code formatter called CODEBUFF that uses machine learning to abstract formatting rules from a representative corpus, using a carefully designed feature set. Our experiments on Java, SQL, and ANTLR grammars show that CODEBUFF is efficient, has excellent accuracy, and is grammar invariant for a given language. It also generalizes to a 4th language tested during manuscript preparation.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding - Pretty printers

Keywords Formatting algorithms, pretty-printer

1. Introduction

The way source code is formatted has a significant impact on its comprehensibility [9], and manually reformatting code is just not an option [8, p.399]. Therefore, programmers need ready access to automatic code formatters or "pretty printers" in situations where formatting is messy or inconsistent. Many program generators also take advantage of code formatters to improve the quality of their output.

Because the value of a particular code formatting style is a subjective notion, often leading to heated discussions, formatters must be highly configurable. This allows, for example, current maintainers of existing code to improve their effectiveness by reformatting the code per their preferred style. There are plenty of configurable formatters for existing languages, whether in IDEs like Eclipse or standalone tools like Gnu `indent`, but specifying style is not easy. The emergent behavior is not always obvious, there exists interdependency between options, and the tools cannot take context information into account [13]. For example, here are the options needed to obtain K&R C style with `indent`:

```
-nbad -bap -bbo -nbc -br -brs -c33 -cd33 -ncdb -ce
-ci4 -cli0 -cp33 -cs -d0 -di1 -nfc1 -nfca -hnl -i4
-ip0 -l75 -lp -npcs -nprs -npsl -saf -sai -saw -nsc
-nsob -nss
```

New languages pop into existence all the time and each one could use a formatter. Unfortunately, building a formatter is difficult and tedious. Most formatters used in practice are *ad hoc*, language-specific programs but there are formal approaches that yield good results with less effort. Rule-based formatting systems let programmers specify phrase-formatting pairs, such as the following sample specification for formatting the COBOL `MOVE` statement using ASF+SDF [3, 12, 13, 15].

```
MOVE IdOrLit TO Id-list =
from-box( H [ "MOVE"
              H ts=25 [to-box(IdOrLit)]
              H ts=49 ["TO"]
              H ts=53 [to-box(Id-list)] ])
```

This rule maps a parse tree pattern to a box expression. A set of such rules, complemented with default behavior for the unspecified parts, generates a single formatter with a specific style for the given language. Section 6 has other related work.

There are a number of problems with rule-based formatters. First, each specification yields a formatter for one specific style. Each new style requires a change to those rules or the creation of a new set. Some systems allow the rules to be parametrized, and configured accordingly, but that leads to higher rule complexity. Second, minimal changes to the associated grammar usually require changes to the format-

ting rules, even if the grammar changes do not affect the language recognized. Finally, formatter specifications are big. Although most specification systems have builtin heuristics for default behavior in the absence of a specification for a given language phrase, specification size tends to grow with the grammar size. A few hundred rules are no exception.

Formatting is a problem solved in theory, but not yet in practice. Building a good code formatter is still too difficult and requires way too much work; we need a fresh approach. In this paper, we introduce a tool called CODEBUFF [11] that uses machine learning to produce a formatter entirely from a grammar for language L and a representative corpus written in L. There is no specification work needed from the user other than to ensure reasonable formatting consistency within the corpus. The statistical model used by CODEBUFF first learns the formatting rules from the corpus, which are then applied to format other documents in the same style. Different corpora effectively result in different formatters. From a user perspective the formatter is "configured by example."

Contributions and roadmap. We begin by showing sample CODEBUFF output in Section 2 and then explain how and why CODEBUFF works in Section 3. Section 4 provides empirical evidence that CODEBUFF learns a formatting style quickly and using very few files. CODEBUFF approximates the corpus style with high accuracy for the languages ANTLR, Java and SQL, and it is largely insensitive to language-preserving grammar changes. To adjust for possible selection bias and model overfitting to these three well-known languages, we tested CODEBUFF on an unfamiliar language (Quorum) in Section 5, from which we learned that CODEBUFF works similarly well, yet improvements are still possible. We position CODEBUFF with respect to the literature on formatting in Section 6.

2. Sample Formatting

This section contains sample SQL, Java, and ANTLR code formatted by CODEBUFF, including some that are poorly formatted to give a balanced presentation. Only the formatting style matters here so we use a small font for space reasons. Github [11] has a snapshot of all input corpora and formatted versions (corpora, testing details in Section 4). To arrive at the formatted output for document d in corpus D, our test rig removes all whitespace tokens from d and then applies an instance of CODEBUFF trained on the corpus without d, $D \setminus \{d\}$.

The examples are not meant to illustrate "good style." They are simply consistent with the style of a specific corpus. In Section 4 we define a metric to measure the success of the automated formatter in an objective and reproducible manner. No quantitative research method can capture the qualitative notion of style, so we start with these examples. (We use "..." for immaterial text removed to shorten samples.)

SQL is notoriously difficult to format, particularly for nested queries, but CODEBUFF does an excellent job in most cases. For example, here is a formatted query from file IP-MonVerificationMaster.sql (trained with `sqlite` grammar on `sqlclean` corpus):

```
SELECT DISTINCT
    t.server_name
       , t.server_id
       , 'Message Queuing Service' AS missingmonitors
FROM t_server t INNER JOIN t_server_type_assoc tsta ON t.server_id = tsta.server_id
WHERE t.active = 1 AND tsta.type_id IN ('8')
      AND t.environment_id = 0
      AND t.server_name NOT IN
    (
         SELECT DISTINCT l.address
         FROM ipmongroups g INNER JOIN ipmongroupmembers m ON g.groupid = m.groupid
              INNER JOIN ipmonmonitors l ON m.monitorid = l.monitorid
              INNER JOIN t_server t ON l.address = t.server_name
              INNER JOIN t_server_type_assoc tsta ON t.server_id = tsta.server_id
         WHERE l.name LIKE '%Message Queuing Service%'
               AND t.environment_id = 0
               AND tsta.type_id IN ('8')
               AND g.groupname IN ('Prod O/S Services')
               AND t.active = 1
     )
UNION
ALL
```

And here is a complicated query from dmart_bits_IAPPBO510.sql with case statements:

```
SELECT
    CASE WHEN SSISInstanceID IS NULL
        THEN 'Total'
    ELSE SSISInstanceID END SSISInstanceID
    , SUM(OldStatus4) AS OldStatus4
    ...
    , SUM(OldStatus4 + Status0 + Status1 + Status2 + Status3 + Status4) AS InstanceTotal
FROM
    (
        SELECT
            CONVERT(VARCHAR, SSISInstanceID)             AS SSISInstanceID
            , COUNT(CASE WHEN Status = 4 AND
                              CONVERT(DATE, LoadReportDBEndDate) <
                              CONVERT(DATE, GETDATE())
                         THEN Status
                    ELSE NULL END)             AS OldStatus4
            ...
            , COUNT(CASE WHEN Status = 4 AND
                              DATEPART(DAY, LoadReportDBEndDate) = DATEPART(DAY, GETDATE())
                         THEN Status
                    ELSE NULL END)             AS Status4
        FROM dbo.ClientConnection
        GROUP BY SSISInstanceID
    ) AS StatusMatrix
GROUP BY SSISInstanceID
```

Here is a snippet from Java, our 2nd test language, taken from STLexer.java (trained with `java` grammar on `st` corpus):

```
switch ( c ) {
    ...
    default:
        if ( c==delimiterStopChar ) {
            consume();
            scanningInsideExpr = false;
            return newToken(RDELIM);
        }
        if ( isIDStartLetter(c) ) {
            ...
            if ( name.equals("if") ) return newToken(IF);
            else if ( name.equals("endif") ) return newToken(ENDIF);
            ...
            return id;
        }
        RecognitionException re = new NoViableAltException("", 0, 0, input);
        ...
        errMgr.lexerError(input.getSourceName(),
                          "invalid character '"+str(c)+"'",
                          templateToken,
                          re);
        ...
```

Here is an example from `STViz.java` that indents a method declaration relative to the start of an expression rather than the first token on the previous line:

```
Thread t = new Thread() {
              @Override
              public void run() {
                  synchronized ( lock ) {
                      while ( viewFrame.isVisible() ) {
                          try {
                              lock.wait();
                          }
                          catch (InterruptedException e) {
                          }
                      }
                  }
              }
          };
```

Formatting results are generally excellent for ANTLR, our third test language. E.g., here is a snippet from Java.g4:

```
classOrInterfaceModifier
    :   annotation       // class or interface
    |   (   'public'     // class or interface
            ...
        |   'final'      // class only -- does not apply to interfaces
        |   'strictfp'   // class or interface
        )
    ;
```

Among the formatted files for the three languages, there are a few regions of inoptimal or bad formatting. CODEBUFF does not capture all formatting rules and occasionally gives puzzling formatting. For example, in the Java8.g4 grammar, the following rule has all elements packed onto one line ("↩" means we soft-wrapped output for printing purposes):

```
unannClassOrInterfaceType
    :   (unannClassType_lfno_unannClassOrInterfaceType | ↩
unannInterfaceType_lfno_unannClassOrInterfaceType) ↩
(unannClassType_lf_unannClassOrInterfaceType |↩
unannInterfaceType_lf_unannClassOrInterfaceType)*
    ;
```

CODEBUFF does not consider line length during training or formatting, instead mimicking the natural line breaks found among phrases of the corpus. For Java and SQL this works very well, but not always with ANTLR grammars.

Here is an interesting Java formatting issue from Compiler.java that is indented too far to the right (column 102); it is indented from the {{. That is a good decision in general, but here the left-hand side of the assignment is very long, which indents the `put()` code too far to be considered good style.

```
public ... Map<...> defaultOptionValues = new HashMap<...>() {{
                                                    put("anchor", "true");
                                                    put("wrap", "\n");
                                                }};
```

In STGroupDir.java, the `prefix` token is aligned improperly:

```
if ( verbose ) System.out.println("loadTemplateFile("+unqualifiedFileName+") in groupdir...
                                  " prefix=" +
prefix);
```

We also note that some of the SQL expressions are incorrectly aligned, as in this sample from SQLQuery23.sql:

```
AND dmcl.ErrorMessage NOT LIKE '%Pre-Execute phase is beginning.  %'
                           AND dmcl.ErrorMessage NOT LIKE '%Prepare for Execute phase...
                                                     AND dmcl.ErrorMessage NOT...
```

Despite a few anomalies, CODEBUFF generally reproduces a corpus' style well. Now we describe the design used to achieve these results. In Section 4 we quantify them.

3. The Design of an AI for Formatting

Our AI formatter mimics what programmers do during the act of entering code. Before entering a program symbol, a programmer decides *(i)* whether a space or line break is required and if line break, *(ii)* how far to indent the next line. Previous approaches (see Section 6) make a language engineer define whitespace injection programmatically.

A formatting engine based upon machine learning operates in two distinct phases: *training* and *formatting*. The *training* phase examines a corpus of code documents, D, written in language L to construct a statistical *model* that represents the formatting style of the corpus author. The essence of training is to capture the whitespace preceding each token, t, and then associate that whitespace with the phrase context surrounding t. Together, the context and whitespace preceding t form an *exemplar*. Intuitively, an exemplar captures how the corpus author formatted a specific, fine-grained piece of a phrase, such as whether the author placed a newline before or after the left curly brace in the context of a Java `if`-statement.

Training captures the context surrounding t as an m-dimensional *feature vector*, X, that includes t's token type, parse-tree ancestors, and many other *features* (Section 3.3). Training captures the whitespace preceding t as the concatenation of two separate operations or *directives*: a whitespace ws directive followed by a horizontal positioning $hpos$ directive if ws is a newline (line break). The ws directive generates spaces, newlines, or nothing while $hpos$ generates spaces to indent or align t relative to a previous token (Section 3.1).

As a final step, training presents the list of exemplars to a machine learning algorithm that constructs a statistical model. There are N exemplars (X_j, w_j, h_j) for $j = 1..N$ where N is the number of total tokens in all documents of corpus D and $w_j \in ws$, $h_j \in hpos$. Machine learning models are typically both a highly-processed condensation of the exemplars and a *classifier function* that, in our case, classifies a context feature vector, X, as needing a specific bit of whitespace. Classifier functions predict how the corpus author would format a specific context by returning a formatting directive. A model for formatting needs two classifier functions, one for predicting ws and one for $hpos$ (consulted if ws prediction yields a newline).

CODEBUFF uses a k-Nearest Neighbor (*kNN*) machine learning model, which conveniently uses the list of exemplars as the actual model. A *kNN*'s classifier function compares an unknown context vector X to the X_j from all N exemplars and finds the k nearest. Among these k, the classifier predicts the formatting directive that appears most often (details in Section 3.4). It's akin to asking a programmer how they normally format the code in a specific situation. Training requires a corpus D written in L, a lexer and parser for L derived from grammar G, and the corpus indentation size to identify indented phrases; e.g., one of the Java corpora we tested indents with 2 not 4 spaces. Let $F_{D,G} = (\mathbf{X}, W, H, \text{indentSize})$ denote the formatting model contents with context vectors forming rows of matrix $\mathbf{X}$ and formatting directives forming elements of vectors W and H. Function 1 embodies the training process, constructing $F_{D,G}$.

Once the model is complete, the *formatting* phase can begin. Formatting operates on a single document d to be formatted and functions with guidance from the model. At each token $t_i \in d$, formatting computes the feature vector X_i representing the context surrounding t_i, just like training does, but does not add X_i to the model. Instead, the formatter presents X_i to the ws classifier and asks it to predict a ws directive for t_i based upon how similar contexts were formatted in the corpus. The formatter "executes" the directive and, if a newline, presents X_i to the $hpos$ classifier to get an indentation

Function 1: $train(D, G, \text{indentSize}) \rightarrow$ model $F_{D,G}$
$\mathbf{X} := []$; $W := []$; $H := []$; $j := 1$;
foreach *document* $d \in D$ **do**
  tokens := tokenize(d);
  tree := parse(*tokens*);
  foreach $t_i \in$ *tokens* **do**
    $\mathbf{X}[j]$:= compute context feature vector for t_i, *tree*;
    $W[j]$:= *capture_ws*(t_i);
    $H[j]$:= *capture_hpos*(t_i, indentSize);
    $j := j + 1$;
  end
end
return (**X**, W, H, indentSize);

or alignment directive. After emitting any preceding whitespace, the formatter emits the text for t_i. Note that any token t_i is identified by its token type, string content, and offset within a specific document, i.

The greedy, "local" decisions made by the formatter give "globally" correct formatting results; selecting features for the X vectors is critical to this success. Unlike typical machine learning tasks, our predictor functions do not yield trivial categories like "it's a cat." Instead, the predicted ws and $hpos$ directives are parametrized. The following sections detail how CODEBUFF captures whitespace, computes feature vectors, predicts directives, and formats documents.

3.1 Capturing whitespace as directives

In order to reproduce a particular style, formatting directives must encode the information necessary to reproduce whitespace encountered in the training corpus. There are five canonical formatting directives:

1. nl: Inject newline
2. sp: Inject space character
3. $(align, t)$: Left align current token with previous token t
4. $(indent, t)$: Indent current token from previous token t
5. $none$: Inject nothing, no indentation, no alignment

For simplicity and efficiency, prediction for nl and sp operations can be merged into a single "predict whitespace" or ws operation and prediction of $align$ and $indent$ can be merged into a single "predict horizontal position" or $hpos$ operation. While the formatting directives are 2- and 3-tuples (details below), we pack the tuples into 32-bit integers for efficiency, w for ws directives and h for $hpos$.

Function 2: *capture_ws*(t_i) $\rightarrow w \in ws$
newlines := num newlines between t_{i-1} *and* t_i*;*
if *newlines* > 0 **then return** $(nl$, *newlines);*
colΔ := t_i.*col* - (t_{i-1}.*col* + *len*(*text*(t_{i-1})));
return $(ws, col\Delta)$*;*

For ws operations, the formatter needs to know how many (n) characters to inject: $ws \in \{(nl, n), (sp, n), none\}$ as shown in Function 2. For example, in the following Java fragment, the proper ws directive at $\uparrow_a$ is $(sp, 1)$, meaning "inject 1 space," the directive at $\uparrow_b$ is $none$, and $\uparrow_c$ is $(nl, 1)$, meaning "inject 1 newline."

```
x␣=␣y;
  ↑a  ↑b
z++;
↑c
```

The $hpos$ directives align or indent token t_i relative to some previous token, t_j for $j < i$ as computed by Function 3. When a suitable t_j is unavailable, there are $hpos$ directives that implicitly align or indent t_i relative to the first token of the previous line:

$hpos \in \{(align, t_j), (indent, t_j), align, indent\}$

In the following Java fragments, assuming 4-space indentation, directive $(indent, \texttt{if})$ captures the whitespace at position $\uparrow_a$, $(align, \texttt{if})$ captures $\uparrow_b$, and $(align, \texttt{x})$ captures $\uparrow_c$.

```
if␣(␣b␣)␣{          f(x,          for (int i=0; ...
␣␣␣␣z++;             ␣␣y)          ␣␣␣␣x=i;
    ↑a                 ↑c              ↑d
}
↑b
```

At position $\uparrow_d$, both $(indent, \texttt{for})$ and $(align, \text{'('})$ capture the formatting, but training chooses indentation over alignment directives when both are available. We experimented with the reverse choice, but found this choice better. Here, $(align, \text{'('})$ inadvertently captures the formatting because `for` happens to be 3 characters.

Function 3: *capture_hpos*(t_i, indentSize) $\rightarrow h \in hpos$
ancestor := leftancestor(t_i)*;*
if $\exists$ *ancestor w/child aligned with* t_i.*col* **then**
  h_{align} := $(align, ancestor\Delta, childindex)$
    with smallest $ancestor\Delta$ & $childindex$*;*
if $\exists$ *ancestor w/child at* t_i.*col* + indentSize **then**
  h_{indent} := $(indent, ancestor\Delta, childindex)$
    with smallest $ancestor\Delta$ & $childindex$*;*
if h_{align} *and* h_{indent} *not nil* **then**
  return *directive with smallest* $ancestor\Delta$*;*
if h_{align} *not nil* **then return** h_{align}*;*
if h_{indent} *not nil* **then return** h_{indent}*;*
if t_i *indented from previous line* **then return** *indent;*
return *align;*

To illustrate the need for $(indent, t_j)$ versus plain $indent$, consider the following Java method fragment where the first statement is not indented from the previous line.

```
public void write(String str)
␣␣␣␣throws IOException {
␣␣␣␣int n = 0;
    ↑
```

Directive $(indent, \texttt{public})$ captures the indentation of `int` but plain $indent$ does not. Plain $indent$ would mean indenting 4 spaces from `throws`, the first token on the previous line, incorrectly indenting `int` 8 spaces relative to `public`.

Directive *indent* is used to approximate nonstandard indentation as in the following fragment.

```
f(100,
␣␣␣0);
   ↑
```

At the indicated position, the whitespace does not represent alignment or standard 4 space indentation. As a default for any nonstandard indentation, function *capture_hpos* returns plain *indent* as an approximation.

When no suitable alignment or indentation token is available, but the current token is aligned with the previous line, training captures the situation with directive *align*:

```
return x +
␣␣␣␣y +
␣␣␣␣z; // align with first token of previous line
```

While $(align, \texttt{y})$ is valid, that directive is not available because of limitations in how *hpos* directives identify previous tokens, as discussed next.

3.2 How Directives Refer to Earlier Tokens

The manner in which training identifies previous tokens for *hpos* directives is critical to successfully formatting documents and is one of the key contributions of this paper. The goal is to define a "token locator" that is as general as possible but that uses the least specific information. The more general the locator, the more previous tokens directives can identify. But, the more specific the locator, the less applicable it is in other contexts. Consider the indicated positions within the following Java fragments where *align* directives must identify the first token of previous function arguments.

```
f(x,          f(x+1,        f(x+1,
␣␣y)          ␣␣y)          ␣␣y,
  ↑a            ↑b          ␣␣␣␣z)
                              ↑c
```

The absolute token index within a document is a completely general locator but is so specific as to be inapplicable to other documents or even other positions within the same document. For example, all positions $\uparrow_a$, $\uparrow_b$, and $\uparrow_c$ could use a single formatting directive, $(align, i)$, but `x`'s absolute index, i, is valid only for a function call at that specific location.

The model also cannot use a relative token index referring backwards. While still fully general, such a locator is still too specific to a particular phrase. At position $\uparrow_a$, token `x` is at delta 2, but at position $\uparrow_b$, `x` is at delta 4. Given argument expressions of arbitrary size, no single token index delta is possible and so such deltas would not be widely applicable. Because the delta values are different, the model could not use a single formatting directive to mean "align with previous argument." The more specific the token locator, the more specific the context information in the feature vectors needs to be, which in turn, requires larger corpora (see Section 3.3).

We have designed a token locator mechanism that strikes a balance between generality and applicability. Not every previous token is reachable but the mechanism yields a single

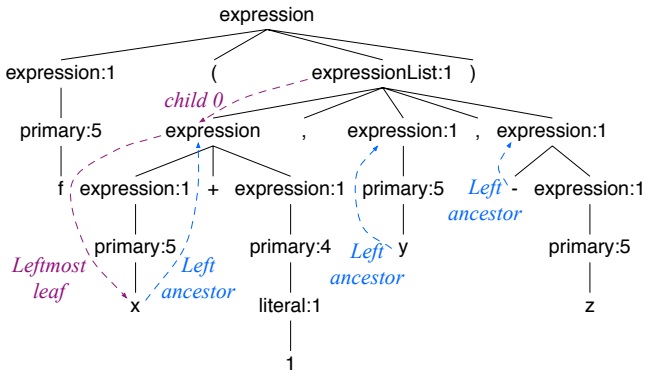


Figure 1. Parse tree for `f(x+1,y,-z)`. Node *rule:n* in the tree indicates the grammar rule and alternative production number used to match the subtree phrase.

locator for `x` from all three positions above and has proven widely applicable in our experiments. The idea is to pop up into the parse tree and then back down to the token of interest, `x`, yielding a locator with two components: A path length to an ancestor node and a child index whose subtree's leftmost leaf is the target token. This alters formatting directives relative to previous tokens to be: $(\_, ancestor\Delta, child)$.

Unfortunately, training can make no assumptions about the structure of the provided grammar and, thus, parse-tree structure. So, training at t_i involves climbing upwards in the tree looking for a suitable ancestor. To avoid the same issues with overly-specific elements that token indexes have, the path length is relative to what we call the *earliest left ancestor* as shown in the parse tree in Figure 1 for `f(x+1,y,-z)`.

The *earliest left ancestor* (or just *left ancestor*) is the oldest ancestor of t whose leftmost leaf is t, and identifies the largest phrase that starts with t. (For the special case where t has no such ancestor, we define left ancestor to be t's parent.) It attempts to answer "what kind of thing we are looking at." For example, the left ancestor computed from the left edge of an arbitrarily-complex expression always refers to the root of the entire expression. In this case, the left ancestors of `x`, `y`, and `z` are siblings, thus, normalizing leaves at three different depths to a common level. The token locator in a directive for `x` in `f(x+1,y,-z)` from both `y` and `z` is $(\_, ancestor\Delta, child) = (\_, 1, 0)$, meaning jump up 1 level from the left ancestor and down to the leftmost leaf of the ancestor's child 0.

The use of the left ancestor and the ancestor's leftmost leaf is critical because it provides a normalization factor among dissimilar parse trees about which training has no inherent structural information. Unfortunately, some tokens are unreachable using purely leftmost leaves. Consider the `return x+y+z;` example from the previous section and one possible parse tree for it in Figure 2. Leaf `y` is unreachable as part of formatting directives for `z` because `y` is not a leftmost leaf of an ancestor of `z`. Function *capture_hpos* must either align or indent relative to `x` or fall back on the plain *align* and *indent*.

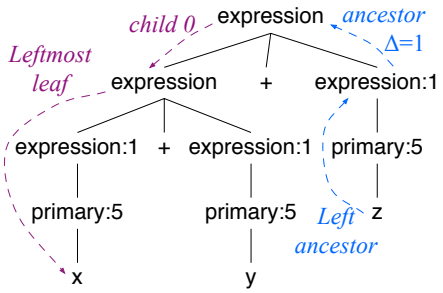


Figure 2. Parse tree for `x+y+z;`.

The opposite situation can also occur, where a given token is unintentionally aligned with or indented from multiple tokens. In this case, training chooses the directive with the smallest $ancestor\Delta$, with ties going to indentation.

And, finally, there could be multiple suitable tokens that share a common ancestor but with different child indexes. For example, if all arguments of `f(x+1,y,-z)` are aligned, the parse tree in Figure 1 shows that $(align, 1, 0)$ is suitable to align `y` and both $(align, 1, 0)$ and $(align, 1, 2)$ could align argument `-z`. Ideally, the formatter would align all function arguments with the same directive to reduce uncertainty in the classifier function (Section 3.4) so training chooses $(align, 1, 0)$ for both function arguments.

The formatting directives capture whitespace in between tokens but training must also record the context in which those directives are valid, as we discuss next.

3.3 Token Context—Feature Vectors

For each token present in the corpus, training computes an exemplar that associates a context with a ws and $hpos$ formatting-directive: (X, w, h). Each context has several features combined into a m-dimensional feature vector, X. The context information captured by the features must be specific enough to distinguish between language phrases requiring different formatting but not so specific that classifier functions cannot recognize any contexts during formatting. The shorter the feature vector, the more situations in which each exemplar applies. Adding more features also has the potential to confuse the classifier.

Through a combination of intuition and exhaustive experimentation, we have arrived at a small set of features that perform well. There are 22 context features computed during training for each token, but ws prediction uses only 11 of them and $hpos$ uses 17. (The classifier function knows which subset to use.) The feature set likely characterises the context needs of the languages we tested during development to some degree, but the features appear to generalize well (Section 5).

Before diving into the feature details, it is worth describing how we arrived at these 21 features and how they affect formatter precision and generality. We initially thought that a sliding window of, say, four tokens would be sufficient context to make the majority of formatting decisions. For example, the context for $\cdots$`x=1*`$\cdots$ would simply be the token types of the surrounding tokens: X=[*id*,=,*int_literal*,*]. The surrounding tokens provide useful but highly-specific information that does not generalize well. Upon seeing this exact sequence during formatting, the classifier function would find an exact match for X in the model and predict the associated formatting directive. But, the classifier would not match context $\cdots$`x=y+`$\cdots$ to the same X, despite having the same formatting needs.

Corpus	N tokens	Unique ws	Unique $hpos$
antlr	19,692	3.0%	4.7%
java	42,032	3.9%	17.4%
java8	42,032	3.4%	7.5%
java_guava	499,029	0.8%	8.1%
sqlite	14,758	8.4%	30.8%
tsql	14,782	7.5%	17.9%

Figure 3. Percentage of unique context vectors in corpora.

The more unique the context, the more specific the formatter can be. Imagine a context for token t_i defined as the 20-token window surrounding each t_i. Each context derived from the corpus would likely be unique and the model would hold a formatting directive specific to each token position of every file. A formatter working from this model could reproduce with high precision a very similar unknown file. The trade-off to such precision is poor generality because the model has "overfit" the training data. The classifier would likely find no exact matches for many contexts, forcing it to predict directives from poorly-matched exemplars.

To get a more general model, context vectors use at most two exact token type but lots of context information from the parse tree (details below). The parse tree provides information about the *kind* of phrase surrounding a token position rather than the specific tokens, which is exactly what is needed to achieve good generality. For example, rather than relying solely on the exact tokens after a `=` token, it is more general to capture the fact that those tokens begin an expression. A useful metric is the percentage of unique context vectors, which we counted for several corpora and show in Figure 3. Given the features described below, there are very few unique context for ws decisions (a few %). The contexts for $hpos$ decisions, however, often have many more unique contexts because ws uses 11-vectors and $hpos$ uses 17-vectors. E.g., our reasonably clean SQL corpus has 31% and 18% unique $hpos$ vectors when trained using SQLite and TSQL grammars, respectively.

For generality, the fewer unique contexts the better, as long as the formatter performs well. At the extreme, a model with just one X context would perform very poorly because all exemplars would be of the form $(X, \_, \_)$. The formatting directive appearing most often in the corpus would be the sole directive returned by the classifier function for any X. The optimal model would have the fewest unique contexts but

Corpus	Ambiguous *ws* directives	Ambiguous *hpos* directives
antlr	42.9%	4.3%
java	29.8%	1.7%
java8	31.6%	3.2%
java_guava	23.5%	2.8%
sqlite_noisy	43.5%	5.0%
sqlite	24.8%	5.5%
tsql_noisy	40.7%	6.3%
tsql	29.3%	6.2%

Figure 4. Percentage of unique context vectors in corpora associated with > 1 formatting directive.

all exemplars with the same context having identical formatting directives. For our corpora, we found that a majority of unique contexts for ws and almost all unique contexts for $hpos$ predict a single formatting directive, as shown in Figure 4. For example, 57.1% of the unique `antlr` corpus contexts are associated with just one ws directive and 95.7% of the unique contexts predict one $hpos$ directive. The higher the ambiguity associated with a single context vector, the higher the uncertainty when predicting formatting decisions during formatting.

The `guava` corpus stands out as having very few unique contexts for ws and among the fewest for $hpos$. This gives a hint that the corpus might be much larger than necessary because the other Java corpora are much smaller and yield good formatting results. Figure 8 shows the effect of corpus size on classifier error rates. The error rate flattens out after training on about 10 to 15 corpus files.

In short, few unique contexts gives an indication of the potential for generality and few ambiguous decisions gives an indication of the model's potential for accuracy. These numbers do not tell the entire story because some contexts are used more frequently than others and those might all predict single directives. Further, a context associated with multiple directives could be 99% one specific directive.

With this perspective in mind, we turn to the details of the individual features. The ws and $hpos$ decisions use a different subset of features but we present all features computed during training together, broken into three logical subsets.

3.3.1 Token type and matching token features

At token index i within each document, context feature-vector X_i contains the following features related to previous tokens in the same document.

1. t_{i-1}, token type of previous token
2. t_i, token type of current token
3. Is t_{i-1} the first token on a line?
4. Is paired token for t_i the *first* on a line?
5. Is paired token for t_i the *last* on a line?

Feature #3 allows the model to distinguish between the following two different ANTLR grammar rule styles at $\uparrow_a$, when t_i=`DIGIT`, using two different contexts.

```
DECIMAL␣:␣DIGIT+␣;
        ↑a      ↑b

DECIMAL
␣␣␣␣:␣␣␣DIGIT+
        ↑a
␣␣␣␣;
    ↑b
```

Exemplars for the two cases are:

(X=[`:`, *RULEREF*, false, ...], w=$(sp, 1)$, h=$none$)
(X'=[`:`, *RULEREF*, true, ...], w'=$(sp, 3)$, h'=$none$)

where *RULEREF* is $type$(`DIGIT`), the token type of rule reference `DIGIT` from the ANTLR meta-grammar. Without feature #3, there would be a single context associated with two different formatting directives.

Features #4 and #5 yield different contexts for common situations related to paired symbols, such as { and }, that require different formatting. For example, at position $\uparrow_b$, the model knows that `:` is the paired previous symbol for `;` (details below) and distinguishes between the styles. On the left, `:` is not the first token on a line whereas `:` does start the line for the case on the right, giving two different exemplars:

(X=[..., false, false], w=$(sp, 1)$, h=$none$)
(X'=[..., true, false], w'=$(nl, 1)$, h'=$(align, :)$)

Those features also allow the model to distinguish between the first two following Java cases where the paired symbol for } is sometimes not at the end of the line in short methods.

```
void reset() {x=0;}
```

```
void reset() {
    x=0;
}
```

```
void reset() {
    x=0;}
```

Without features #4-#5, the formatter would yield the third.

Determining the set of paired symbols is nontrivial, given that the training can make no assumptions about the language it is formatting. We designed an algorithm, *pairs* in Function 4, that analyzes the parse trees for all documents in the corpus and computes plausible token pairs for every non-leaf node (grammar rule production) encountered. The algorithm relies on the idea that paired tokens are token literals, occur as siblings, and are not repeated siblings. Grammar authors also do not split paired token references across productions. Instead, authors write productions such as these ANTLR rules for Java:

```
expr : ID '[' expr ']' | ... ;
type : ID '<' ID (',' ID)* '>' | ... ;
```

that yield subtrees with the square and angle brackets as direct children of the relevant production. Repeated tokens are not plausible pair elements so the commas in a generic Java type list, as in `T<A,B,C>`, would not appear in pairs associated with rule `type`. A single subtree in the corpus with repeated commas as children of a `type` node would remove comma from all pairs associated with rule `type`. Further details are available in Function 4 (source CollectTokenPairs.java). The

algorithm neatly identifies pairs such as (`?`, `:`) and (`[`, `]`), and (`(`, `)`) for Java expressions and (`enum`, `}`), (`enum`, `{`), and (`{`, `}`) for enumerated type declarations. During formatting, *paired* (Function 5) returns the paired symbols for t_i.

Function 4: $pairs$(Corpus D) $\rightarrow$ map node $\mapsto$ set<(s,t)>
pairs := map of node $\mapsto$ set<tuples>;
repeats := map of node $\mapsto$ set<token types>;
foreach $d \in D$ **do**
  foreach *non-leaf node* r *in parse(*d*)* **do**
    literals := $\{t \mid parent(t) = r, t \text{ is literal token}\}$;
    add $\{(t_i,t_j) \mid i < j \,\forall\, t_i, t_j \in literals\}$ to *pairs*[r];
    add $\{t_i \mid \exists\, t_i = t_j, i \neq j\}$ to *repeats*[r];
  end
end
delete pair $(t_i,t_j) \in pairs[r]$ if t_i or $t_j \in repeats[r] \,\forall\, r$;
return *pairs*;

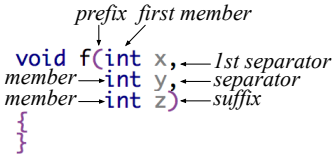
Function 5: $paired$(*pairs*, token t_i) $\rightarrow t'$
mypairs := *pairs*[$parent(t)$];
viable := $\{s \mid (s,t) \in mypairs, s \in siblings(t)\}$;
if $|viable| = 1$ **then** *ttype* := *viable*[0];
else if $\exists(s,t)|$ s,t *are common pairs* **then** *ttype* := s;
else if $\exists(s,t)|$ s,t *are single-char literals* **then** *ttype* := s;
else *ttype* := *viable*[0]; // choose first if still ambiguous
matching := $[t_j \mid t_j = ttype, j < i, \forall\, t_j \in siblings(t_i)]$;
return *last(matching)*;

3.3.2 List membership features

Most computer languages have lists of repeated elements separated or terminated by a token literal, such as statement lists, formal parameter lists, and table column lists. The next group of features indicates whether t_i is a component of a list construct and whether or not that list is split across multiple lines ("oversize").

6. Is $leftancestor(t_i)$ a component of an oversize list?
7. $leftancestor(t_i)$ component type within list from {*prefix token, first member, first separator, member, separator, suffix token*}

With these two features, context vectors capture not only two different overall styles for short and oversize lists but how the various elements are formatted within those two kinds of lists. Here is a sample oversize Java formal parameter list annotated with list component types:



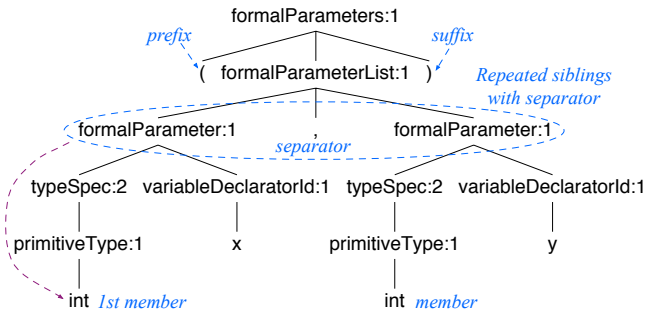


Figure 5. Formal Args Parse Tree `void f(int x, int y)`.

Only the first member of a list is differentiated; all other members are labeled as just plain members because their formatting is typically the same. The exemplars would be:

(X=[..., true, *prefix*], w=$none$, h=$none$)
(X=[..., true, *first member*], w=$none$, h=$none$)
(X=[..., true, *first separator*], w=$none$, h=$none$)
(X=[..., true, *member*], w=$(nl,1)$, h=$(align,$ first arg$)$)
(X=[..., true, *separator*], w=$none$, h=$none$)
(X=[..., true, *member*], w=$(nl,1)$, h=$(align,$ first arg$)$)
(X=[..., true, *suffix*], w=$none$, h=$none$)

Even for short lists on one line, being able to differentiate between list components lets training capture different but equally valid styles. For example, some ANTLR grammar authors write short parenthesized subrules like `(ID|INT|FLOAT)` but some write `(ID | INT | FLOAT)`.

As with identifying token pairs, CODEBUFF must identify the constituent components of lists without making assumptions about grammars that hinder generalization. The intuition is that lists are repeated sibling subtrees with a single token literal between the 1st and 2nd repeated sibling, as shown in Figure 5. Repeated subtrees without separators are not considered lists. Training performs a preprocessing pass over the parse tree for each document, tagging the tokens identified as list components with values for features #6- #7. Tokens starting list members are identified as the leftmost leaves of repeated siblings (`formalParameter` in Figure 5). Prefix and suffix components are the tokens immediately to the left and right of list members but only if they share a common parent.

The training preprocessing pass also collects statistics about the distribution of list text lengths (without whitespace) of regular and oversize lists. Regular and oversize list lengths are tracked per (r,c,sep) combination for rule subtree root type r, child node type c, and separator token type sep; e.g., (r,c,sep)=(`formalParameterList`, `formalParameter`,`','`) in Figure 5. The separator is part of the tuple so that expressions can distinguish between different operators such as `=` and `*`. Children of binary and ternary operator subtrees satisfy the conditions for being a list, with the operator as separator token(s). For each (r,c,sep) combination, training tracks the number of those lists and the median list length, $(r,c,sep) \mapsto (n, median)$.

3.3.3 Identifying oversize lists during formatting

As with training, the formatter performs a preprocessing pass to identify the tokens of list phrases. Whereas training identifies oversize lists simply as those split across lines, formatting sees documents with all whitespace squeezed out. For each (r, c, sep) encountered during the preprocessing pass, the formatter consults a mini-classifier to predict whether that list is oversize or not based upon the list string length, ll. The mini-classifier compares the mean-squared-distance of ll to the median for regular lists and the median for oversize (big) lists and then adjusts those distances according to the likelihood of regular vs oversize lists. The *a priori* likelihood that a list is regular is $p(reg) = n_{reg}/(n_{reg} + n_{big})$, giving an adjusted distance to the regular type list as: $dist_{reg} = (ll - median_{reg})^2 * (1 - p(reg))$. The distance for oversize lists is analogous.

When a list length is somewhere between the two medians, the relative likelihoods of occurrence shift the balance. When there are roughly equal numbers of regular and oversize lists, the likelihood term effectively drops out, giving just mean-squared-distance as the mini-classifier criterion. At the extreme, when all (r, c, sep) lists are big, $p(big) = 1$, forcing $dist_{big}$ to 0 and, thus, always predicting oversize.

When a single token t_i is a member of multiple lists, training and formatting associate t_i with the longest list subphrase because that yields the best formatting, as evaluated manually across the corpora. For example, the expressions within a Java function call argument list are often themselves lists. In `f(`e_1`,...,a+b)`, token `a` is both a sibling of `f`'s argument list but also the first sibling of expression `a+b`, which is also a list. Training and formatting identify `a` as being part of the larger argument list rather than the smaller `a+b`. This choice ensures that oversize lists are properly split. Consider the opposite choice where `a` is associated with list `a+b`. In an oversize argument list, the formatter would not inject a newline before `a`, yielding poor results:

```
f(e1,
  ..., a+b)
```

Because list membership identification occurs in a top-down parse-tree pass, associating tokens with the largest construct is a matter of latching the first list association discovered.

3.3.4 Parse-tree context features

The final features provide parse-tree context information:

8. $childindex(t_i)$
9. $rightancestor(t_{i-1})$
10. $leftancestor(t_i)$
11. $childindex(leftancestor(t_i))$
12. $parent_1(leftancestor(t_i))$
13. $childindex(parent_1(leftancestor(t_i)))$
14. $parent_2(leftancestor(t_i))$
15. $childindex(parent_2(leftancestor(t_i)))$
16. $parent_3(leftancestor(t_i))$
17. $childindex(parent_3(leftancestor(t_i)))$
18. $parent_4(leftancestor(t_i))$
19. $childindex(parent_4(leftancestor(t_i)))$
20. $parent_5(leftancestor(t_i))$
21. $childindex(parent_5(leftancestor(t_i)))$

Here the $childindex(p)$ is the 0-based index of node p among children of $parent(p)$, $childindex(t_i)$ is shorthand for $childindex(leaf(t_i))$, and $leaf(t_i)$ is the leaf node associated with t_i. Function $childindex(p)$ has a special case when p is a repeated sibling. If p is the first element, $childindex(p)$ is the actual child index of p within the children of $parent(p)$ but is special marker * for all other repeated siblings. The purpose is to avoided over-specializing the context vectors to improve generality. These features also use function $parent_i(p)$, which is the i^{th} parent of p; $parent_1(p)$ is synonymous with the direct parent $parent(p)$.

The child index of t_i, feature #8, gives the necessary context information to distinguish the alignment token between the following two ANTLR lexical rules at the semicolon.

```
BooleanLiteral          fragment
    :   'true'          DIGIT
    |   'false'             :   [0-9]
    ;                       ;
```

On the left, the `;` token is child index 3 but 4 on the right, yielding different contexts, X and X', to support different alignment directives for the two cases. Training collects exemplars $(X, (align, 0, 1))$ and $(X', (align, 0, 2))$, which aligns `;` with the colon in both cases.

Next, features $rightancestor(t_{i-1})$ and $leftancestor(t_i)$ describe what phrase precedes t_i and what phrase t_i starts. The $rightancestor$ is analogous to $leftancestor$ and is the oldest ancestor of t_i whose rightmost leaf is t_i (or $parent(t_i)$ if there is no such ancestor). For example, at t_i=`y` in `x=1; y=2;` the right ancestor of t_{i-1} and the left ancestor of t_i are both "statement" subtree roots.

Finally, the parent and child index features capture context information about highly nested constructs, such as:

```
if ( x ) { }
else if ( y ) { }
else if ( z ) { }
else { }
```

Each `else` token requires a different formatting directive for alignment, as shown in Figure 6; e.g., $(align, 1, 3)$ means "jump up 1 level from $leftancestor(t_i)$ and align with leftmost leaf of child 3 (token `else`)." To distinguish the cases, the context vectors must be different. Therefore, training collects these partial vectors with features #10-15:

X=[..., *stat*, 0, *blockStat*, *, `block`, 0, ...]
X=[..., *stat*, *, *stat*, 0, *blockStat*, *, ...]
X=[..., *stat*, *, *stat*, *, *stat*, 0, ...]

where *stat* abbreviates `statement:3` and *blockStat* abbreviates `blockStatement:2`. All deeper `else` clauses also use directive $(align, 1, 3)$.

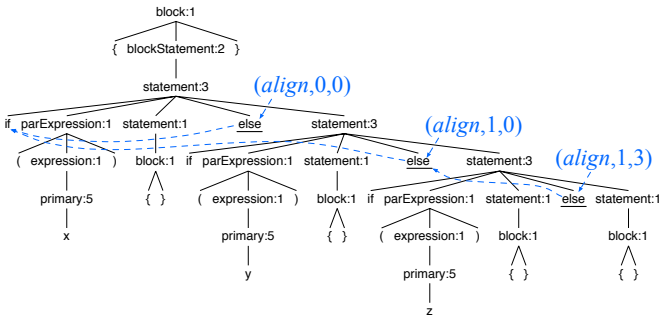


Figure 6. Alignment directives for nested `if-else` statements.

Training is complete once the software has computed an exemplar for each token in all corpus files. The formatting model is the collection of those exemplars and an associated classifier that predicts directives given a feature vector.

3.4 Predicting Formatting Directives

CODEBUFF's *kNN* classifier uses a fixed $k = 11$ (chosen experimentally in Section 4) and an L_0 distance function (ratio of number of components that differ to vector length) but with a twist on classic *kNN* that accentuates feature vector distances in a nonlinear fashion. To make predictions, a classic *kNN* classifier computes the distance from unknown feature vector X to every X_j vector in the exemplars, $(\mathbf{X}, Y)$, and predicts the category, y, occurring most frequently among the k exemplars nearest X.

The classic approach works very well in Euclidean space with quantitative feature vectors but not so well with an L_0 distance that measures how similar two code-phrase contexts are. As the L_0 distance increases, the similarity of two context vectors drops off dramatically. Changing even one feature, such as earliest left ancestor (kind of phrase), can mean very different contexts. This quick drop off matters when counting votes within the k nearest X_j. At the extreme, there could be one exemplar where $X = X_j$ at distance 0 and 10 exemplars at distance 1.0, the maximum distance. Clearly the one exact match should outweigh 10 that do not match at all, but a classic *kNN* uses a simple unweighted count of exemplars per category (10 out of 11 in this case). Instead of counting the number of exemplars per category, our variation sums $1 - \sqrt[3]{L_0(X, X_j)}$ for each X_j per category. Because distances are in $[0..1]$, the cube root nonlinearly accentuates differences. Distances of 0 count as weight 1, like the classic *kNN*, but distances close to 1.0 count very little towards their associated category. In practice, we found feature vectors more distant than about 15% from unknown X to be too dissimilar to count. Exemplars at distances above this threshold are discarded while collecting the k nearest neighbors.

The classfier function uses features #1-#10, #12 to make ws predictions and #2, #6-#21 for $hpos$; $hpos$ predictions ignore X_j not associated with tokens starting a line.

3.5 Formatting a Document

To format document d, the formatter Function 6 first squeezes out all whitespace tokens and line/column information from the tokens of d and then iterates through d's remaining tokens, deciding what whitespace to inject before each token. At each token, the formatter computes a feature vector for that context and asks the model to predict a formatting directive (whereas training examines the whitespace to determine the directive). The formatter uses the information in the formatting directive to compute the number of newline and space characters to inject. The formatter treats the directives like bytecode instructions for a simple virtual machine: $\{(nl, n),\ (sp, n),\ none,\ (align, ancestor\Delta, child),\ (indent, ancestor\Delta, child),\ align,\ indent\}$.

As the formatter emits tokens and injects whitespace, it tracks line and column information so that it can annotate tokens with this information. Computing features #3-5 at token t_i relies on line and column information for t_j for some $j < i$. For example, feature #3 answers whether t_{i-1} is the first token on the line, which requires line and column information for t_{i-1} and t_{i-2}. Because of this, predicting the whitespace preceding token t_i is a (fast) function of the actions made previously by the formatter. After processing t_i, the file is formatted up to and including t_i.

Before emitting whitespace in front of token t_i, the formatter emits any comments found in the source code. (The ANTLR parser has comments available on a "hidden channel".) To get the best output, the formatter needs whitespace in front of comments and this is the one case where the formatter looks at the original file's whitespace. Otherwise, the formatter computes all whitespace generated in between tokens. To ensure single-line comments are followed by a newline, users of CODEBUFF can specify the token type for single-line comments as a failsafe.

4. Empirical results

The primary question when evaluating a code formatter is whether it consistently produces high quality output, and we begin by showing experimentally that CODEBUFF does so. Next, we investigate the key factors that influence CODEBUFF's statistical model and, indirectly, formatting quality: the way a grammar describes a language, corpus size/consistency, and parameter k of the *kNN* model. We finish with a discussion of CODEBUFF's complexity and performance.

4.1 Research Method: Quantifying formatting quality

We need to accurately quantify code formatter quality without human evaluation. A metric helps to isolate issues with the model (and subsequently improve it) as well as report its efficacy in an objective manner. We propose the following measure. Given corpus D that is perfectly consistently formatted, CODEBUFF should produce the identity transformation for any document $d \in D$ if trained on a corpus subset $D \setminus \{d\}$. This *leave-one-out cross-validation* allows us to

Function 6: *format*($F_{D,G} = (\mathbf{X}, W, H, \text{indentSize}), d$)
line := *col* := 0;
d := d with whitespace tokens, line/column info removed;
foreach $t_i \in d$ **do**
  emit any comments to left of t_i;
  X_i := compute context feature vector at t_i;
  ws := predict directive using X_i and $\mathbf{X}$, W;
  newlines := *sp* := 0;
  if $ws = (nl, n)$ **then** *newlines* := n;
  else if $ws = (sp, n)$ **then** *sp* := n;
  if *newlines* > 0 **then** // inject newline and align/indent
    emit $newlines$ '\n' characters;
    line += $newlines$; *col* := 0;
    $hpos$:= predict directive using X_i and $\mathbf{X}$, H;
    if $hpos = (\_, ancestor\Delta, child)$ **then**
      t_j = token relative to t_i at $ancestor\Delta, child$;
      col := t_j.col;
      if $hpos = (indent, \_, \_)$ **then** *col* += indentSize;
      emit *col* spaces;
    else // plain align or indent
      t_j := first token on previous line;
      col := t_j.col;
      if $hpos = indent$ **then** *col* += $indentSize$;
      emit *col* spaces;
    end
  else
    col += *sp*;
    emit *sp* spaces; // inject spaces
  end
  t_i.line = *line*; // set t_i location
  t_i.col = *col*;
  emit $text(t_i)$;
  col += $len(text(t_i))$

use the corpus for both training and for measuring formatter quality. (See Section 5 for evidence of CODEBUFF's generality.) For each document, the distance between original d and formatted d' is an inverse measure of formatting quality.

A naive similarity measure is the edit distance (Levenshtein Distance [7]) between d' and d, but it is expensive to compute and will over-accentuate minor differences. For example, a single indentation error made by the formatter could mean the entire file is shifted too far to the right, yielding a very high edit distance. A human reviewer would likely consider that a small error, given that the code looks exactly right except for the indentation level. Instead, we quantify the document similarity using the aggregate misclassification rate, in $[0..1]$, for all predictions made while generating d':

$$error = \frac{n\_ws\_errors + n\_hpos\_errors}{n\_ws\_decisions + n\_hpos\_decisions}$$

A misclassification error occurs when the *kNN* model predicts a formatting directive for d' at token t_i that differs from the actual formatting found in the original d at t_i. The formatter predicts whitespace for each t_i so $n\_ws\_decisions = |d| = |d'|$, the number of real tokens in d. For each $ws = (nl, \_)$ prediction, the formatter predicts $hpos$ so $n\_hpos\_decisions \leq |d|$. An error rate of 0 indicates that d' is identical to d and an error rate of 1 indicates that every prediction made during formatting of d' would yield formatting that differs from that found in d. Formatting directives that differ solely in the number of spaces or in the relative token identifier count as misclassifications; e.g., $(sp, 1) \neq (sp, 2)$ and $(align, i, j) \neq (align, i', j')$. We consider this error rate an acceptable proxy for human opinion, albeit imperfect.

4.2 Corpora

We selected three very different languages—ANTLR grammars, Java, and SQL—and used the following corpora (stored in CODEBUFF's [11] `corpus` directory).

- `antlr`. A subset of 12 grammars from **ANTLR**'s grammar repository, manually formatted by us.
- `st`. All 59 **Java** source files for StringTemplate.
- `guava`. All 511 **Java** source files for Google's Guava.
- `sql_noisy`. 36 **SQL** files taken from a `github` repository.[1] The SQL corpus was groomed and truncated so it was acceptable to both SQLite and TSQL grammars.
- `sql`. The same 36 **SQL** files as formatted using Intellij IDE; some manual formatting interventions were done to fix Intellij formatting errors.

As part of our examination of grammar invariance (details below), we used two different Java grammars and two different SQL grammars taken from ANTLR's grammar repository:

- `java`. A Java 7 grammar.
- `java8`. A transcription of the Java 8 language specification into ANTLR format.
- `sqlite`. A grammar for the SQLite variant of SQL.
- `tsql`. A grammar for the Transact-SQL variant of SQL.

4.3 Formatting quality results

Our first experiment demonstrates that CODEBUFF can faithfully reproduce the style found in a consistent corpus. Details to reproduce all results are available in a `README.md` [11]. Figure 7 shows the formatting error rate, as described above. Lower median error rates correspond with higher-quality formatting, meaning that the formatted files are closer to the original. Manual inspection of the corpora confirms that consistently-formatted corpora indeed yield better results. For example, median error rates (17% and 19%) are higher using the two grammars on the `sql_noisy` corpus versus the cleaned up `sql` corpus. The `guava` corpus has extremely consistent style because it is enforced programmatically and consequently CODEBUFF is able to reproduce the style with

[1] `https://github.com/mmessano/SQL`

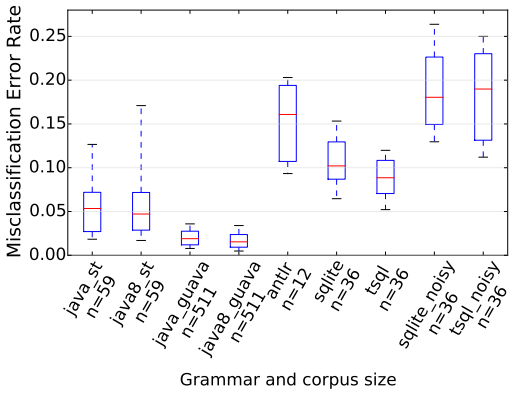


Figure 7. Standard box-plot of leave-one-out validation error rate between formatted document d' and original d.

high accuracy using either Java grammar. The `antlr` corpus results have a high error rate due to some inconsistencies among the grammars but, nonetheless, formatted grammars look good except for a few overly-long lines.

4.4 Grammar invariance

Figure 7 also gives a strong hint that CODEBUFF is *grammar invariant*, meaning that training models on a single corpus but with different grammars gives roughly the same formatting results. For example, the error rates for the `st` corpus trained with `java` and `java8` grammars are roughly the same, indicating that CODEBUFF's overall error rate (similarity of original/formatted documents) does not change when we swap out the grammar. The same evidence appears for the other corpora and grammars. The overall error rate could hide large variation in the formatting of individual files, however, so we define grammar invariance as a file-by-file comparison of normalized edit distances.

Definition 4.1. *Given models $F_{D,G}$ and $F_{D,G'}$ derived from grammars G and G' for a single language, $L(G) = L(G')$, a formatter is* grammar invariant *if the following holds for any document d: $format(F_{D,G}, d) \ominus format(F_{D,G'}, d) \leq \epsilon$ for some suitably small* normalized edit distance ϵ.

Definition 4.2. *Let operator $d_1 \ominus d_2$ be the* normalized edit distance *between documents d_1 and d_2 defined by the Levenshtein Distance [7] divided by $max(len(d_1), len(d_2))$.*

The median edit distances between formatted files (using leave-one-out validation) from 3 corpora provide strong evidence of grammar invariance for Java but less so for SQL:

- 0.001 for `guava` corpus with `java` and `java8` grammars
- 0.008 for `st` corpus with `java` and `java8` grammars
- 0.099 for `sql` corpus with `sqlite` and `tsql` grammars

The "average" difference between `guava` files formatted with different Java grammars is 1 character edit per 1000 characters. The less consistent `st` corpus yields a distance of 8 edits per 1000 characters. A manual inspection of Java documents formatted using models trained with different grammars confirms that the structure of the grammar itself has little to no effect on the formatting results, at least when trained on the context features defined in 3.3.

The `sql` corpus shows a much higher difference between formatted files, 99 edits per 1000 characters. Manual inspection shows that both versions are plausible, just a bit different in nl prediction. Newlines trigger indentation, leading to bigger whitespace differences. One reason for higher edit distances could be that the noise in the less consistent SQL corpus amplifies any effect that the grammar has on formatting. More likely, the increased grammar sensitivity for SQL has to do with the fact that the `sqlite` and `tsql` grammars are actually for two different languages. The TSQL language has procedural extensions and is Turing complete; the `tsql` grammar is 2.5x bigger than `sqlite`. In light of the different SQL dialects and noisier corpus, a larger difference between formatted SQL files is unsurprising and does not rule out grammar invariance.

4.5 Effects of corpus size

Prospective users of CODEBUFF will ask how the size of the corpus affects formatting quality. We performed an experiment to determine: *(i)* how many files are needed to reach the median overall error rate and *(ii)* whether adding more and more files confuses the *kNN* classifier. Figure 8 summarizes the results of an experiment comparing the median error rate for randomly-selected corpus subsets of varying sizes across different corpora and grammars. Each data point represents 50 trials at a specific corpus size. The error rate quickly drops after about 5 files and then asymptotically approaches the median error rate shown in Figure 7. This graph suggests a minimum corpus size of about 10 files and provides evidence that adding more (consistently formatted) files neither confuses the classifier nor improves it significantly.

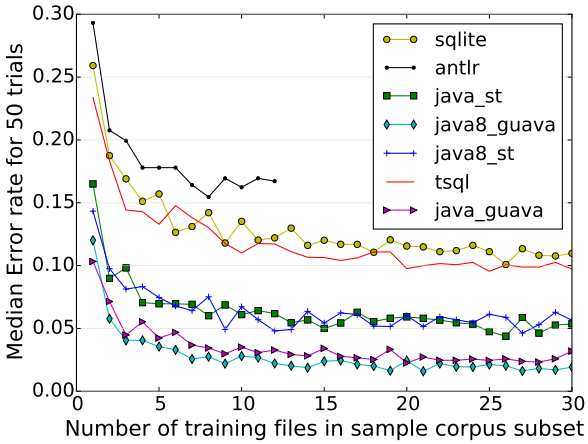


Figure 8. Effect of corpus size on median leave-one-out validation error rate using randomly-selected corpus subsets.

4.6 Optimization and stability of model parameters

Choosing a k for a *kNN* model is more of an art but $k = \sqrt{N}$ for N exemplars is commonly used. Through exhaustive manual review of formatted files, we instead arrived at a fixed $k = 11$ and then verified its suitability by experiment. Figure 9 shows the effect of varying k on the median error rate across a selection of corpora and grammars; k ranged from 1 to 99. This graph supports the conclusion that a formatter can make accurate decisions by comparing the context surrounding t_i to very few model exemplars. Moreover, formatter accuracy is very stable with respect to k; even large changes in k do not alter the error rate very much.

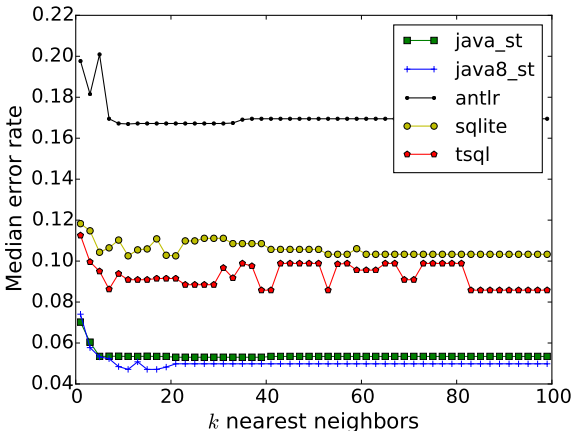


Figure 9. Effect of k on median leave-one-out error rate.

4.7 Worst-case complexity

Collecting all exemplars to train our *kNN* $F_{D,G}$ model requires two passes over the input. The first pass walks the parse tree for each $d \in D$, collecting matching token pairs (Section 3.3.1) and identifying list membership (Section 3.3.2). The size of a single parse tree is bounded by the size of the grammar times the number of tokens in the document, $|G| \times |d|$ for document d (the charge per token is a tree depth of at most $|P|$ productions of G). To make a pass over all parse trees for the entire corpus, the time complexity is $|G| \times N$, so in $O(N)$ for N total tokens in the corpus.

The second pass walks each token $t_i \in d$ for all $d \in D$, using information computed in the first pass to compute feature vectors and capture whitespace as ws and $hpos$ formatting directives. There are m features to compute for each of N tokens. Most of the features require constant time, but computing the earliest ancestors and identifying tokens for indentation and alignment might walk the depth of the parse tree in the worst case for a cost of $O(\log(|G| \times |d|))$ per feature per token. For an average document size, a feature costs $O(\log(|G| \times N/|D|))$. Computing all m features and capturing whitespace for all documents costs

$$O(m \times N \times \log(|G| \times N/|D|)) =$$

$$O(N \times m \times (\log(|G|) + \log(N) - \log(|D|)))$$

which is in $O(N \log N)$. Including the first pass over all parse trees adds a factor of N but does not change the overall worst-case time complexity for training.

Formatting a document is much more expensive than training. For each token in a document, the formatter requires at least one ws classifier function execution and possibly a second for $hpos$. Each classifier function execution requires the feature vector X computation cost per the above, but the cost is dominated by the comparison of X to every $X_j \in \mathbf{X}$ and sorting those results by distance to find the k nearest neighbors. For $n = |d|$ tokens in a file to be formatted, the overall complexity is quadratic, $O(n \times N \log N)$. For an average document size of $n = N/|D|$, formatting a document costs $O(N^2 \log N)$. Box formatters are also usually quadratic; streaming formatters are linear (See Section 6).

4.8 Expected performance

CODEBUFF is instrumented to report single-threaded CPU run-time for training and formatting time. We report a median $1.5s$ training time for the (overly-large by an order magnitude) `guava` corpus (511 files, 143k lines) after parsing documents using the `java` grammar and a training time of $1.8s$ with the `java8` grammar.[2] The `antlr` corpus, with only 12 files, is trained within $72ms$.

Because *kNN* uses the complete list of exemplars as an internal representation, the memory footprint of CODEBUFF is substantial. For each of N tokens in the corpus, we track m features and two formatting directives (each packed into a single word). The size complexity is $O(N)$ but with a nontrivial constant. For the `guava` corpus, the profiler shows 2.5M $X_j \in \mathbf{X}$ feature vectors, consuming 275M RAM. Because CODEBUFF keeps all corpus text in memory for testing purposes, we observe an overall 1.7G RAM footprint.

For even a modest sized corpus, the number of tokens, N, is large and a naive *kNN* classifier function implementation is unusably slow. Ours uses an index to narrow the nearest neighbors search space and a cache of prediction function results to make the performance acceptable. To provide worst-case formatting speed figures, CODEBUFF formats the biggest file (LocalCache.java) in the `guava` library (22,674 tokens, 5022 lines) in median times of $2.7s$ (`java`) and $2.2s$ (`java8`). At ~2300 lines/s even for such a large training corpus, CODEBUFF is usable in IDEs and other applications. Using the more reasonably-sized `antlr` corpus, formatting the 1777 line `Java8.g4` takes just $70ms$ (25k lines/s).

5. Test of Generality

As a test of generality, we solicited a grammar and corpus for an unfamiliar language, Quorum[3] by Andreas Stefik,

[2] Experiments run 20 times on an iMac17,1 OS 10.11.5, 4GHz Intel Core i7 with Java 8, heap/stack size 4G/1M; we ignore first 5 measurements to account for warmup time. Details to reproduce in github repo.

[3] `https://www.quorumlanguage.com`

and trained CODEBUFF on that corpus. The first look at the grammar and corpus for both the authors and the model occurred during the preparation of this manuscript. We dropped about half of the corpus files randomly to simulate a more reasonably-sized training corpus but did not modify any files.

Most files in the corpus result in plausibly formatted code, but there are a few outlier files, such as HashTable.quorum. For example, the following function looks fine except that it is indented way too far to the right in the formatted output:

```
action RemoveKey(Key key) returns Value
    ...
    repeat while node not= undefined
        if key:Equals(node:key)
            if previous not= undefined
                previous:next = node:next
            else
                array:Set(index, node:next)
            end
        ...
    end
    return undefined
end
```

The median misclassification error rate for Quorum is very low, 4%, on a par with the `guava` model, indicating that it is consistent and that CODEBUFF captures the formatting style.

6. Related work

The state of the art in language-parametric formatting stems from two sources: Coutaz [3] and Oppen [10]. Coutaz introduced the "Box" as a basic two-dimensional layout mechanism and all derived work elaborates on specifying the mapping from syntax (parse) trees to Box expressions more effectively (in terms of meta-program size and expressiveness) [4, 12–14]. Oppen introduced the streaming formatting algorithm in which alignment and indentation operators are injected into a token stream (coming from a syntax tree serialization or a lexer token stream). Derivative and later work [1, 2, 18] elaborates on specifying how and when to inject these tokens for different languages in different ways and in different technological spaces [6].

Oppen's streams shine for their simplicity and efficiency in time and memory, while the expressivity of Coutaz' Boxes makes it easier to specify arbitrary formatting styles. The expressivity comes at the cost of building an intermediate representation and running a specialized constraint solver.

Conceptually CODEBUFF derives from the Oppen school most, but in terms of expressivity and being able to capture "naturally occurring" and therefore hard to formalize formatting styles, CODEBUFF approaches the power of Coutaz' Boxes. The reason is that CODEBUFF matches token context much like the algorithms that map syntax trees to Box expressions. CODEBUFF can, therefore, seamlessly learn to *specialize* for even more specific situations than a software language engineer would care to express. At the same time, it uses an algorithm that simply injects layout directives into a token stream, which is very efficient. There are limitations, however (detailed in Section 3).

Language-parametric formatters from the Box school are usually combined with *default formatters*, which are statically constructed using heuristics (expert knowledge). The default formatter is expected to guess the right formatting rule in order to relieve the language engineer from having to specify a rule for every language construct (like CODEBUFF does for all constructs). To do this, the input grammar is analyzed for "typical constructs" (such as expressions and block constructs), which are typically formatted in a particular way. The usefulness of default formatters is limited though. We have observed engineers mapping languages to Box completely manually, while a default formatter was readily available. If CODEBUFF is not perfect, at least it generalizes the concept of a default formatter by learning from input sentences and not just the grammar of a language. This difference is the key benefit of the machine learning approach; CODEBUFF could act as a more intelligent default formatter in existing formatting pipelines.

PGF [1] by Bagge and Hasu is a rule-based system to inject pretty printing directives into a stream. The meta-grammar for these rules (which the language engineer must write) is conceptually comparable to the feature set that CODEBUFF learns. The details of the features are different however, and these details matter greatly to the efficacy of the learner. The interplay between the expressiveness of a meta-grammar for formatting rules and the features CODEBUFF uses during training is interesting to observe: we are characterizing the domain of formatting from different ends.

Since CODEBUFF automatically learns by example, related work that needs explicit formalizations is basically incomparable from a language engineering perspective. We can only compare the speed of the derived formatters and the quality of the formatted output. It would be possible to compare the default grammar-based formatters to see how many of their rules are useful compared to CODEBUFF, but *(i)* this is completely unfair because the default formatters do not have access to input examples and *(ii)* default formatters are constructed in a highly arbitrary manner so there is no lesson to learn other than their respective designer's intuitions.

Aside from the actual functionality of formatting text, related work has extended formatting with "high fidelity" (not losing comments) [16, 17] and partial formatting (introducing a new source text in an already formatted context) [5]. CODEBUFF copies comments into the formatted output but partial formatting is outside the scope of the current contribution; CODEBUFF could be extended with such functionality.

7. Conclusion

CODEBUFF is a step towards a universal code formatter that uses machine learning to abstract formatting rules from a representative corpus. Current approaches require complex pattern-formatting rules written by a language expert whereas input to CODEBUFF is just a grammar, corpus, and indentation size. Experiments show that training requires about 10

files and that resulting formatters are fast and highly accurate for three languages. Further, tests on a previously-unseen language and corpus show that, without modification, CODEBUFF generalized to a 4th language. Formatting results are largely insensitive to language-preserving changes in the grammar and our *kNN* classifier is highly stable with respect to changes in model parameter k. Based on these results, we look forward to many more applications of machine learning in software language engineering.

References

[1] A. H. Bagge and T. Hasu. A pretty good formatting pipeline. In *International Conferance on Software Language Engineering (SLE'13)*, volume 8225 of *LNCS*. Springer, Oct. 2013.

[2] O. Chitil. Pretty printing with lazy dequeues. *ACM TOPLAS*, 27(1):163–184, Jan. 2005. ISSN 0164-0925.

[3] J. Coutaz. The Box, a layout abstraction for user interface toolkits. CMU-CS-84-167, Carnegie Mellon University, 1984.

[4] M. de Jonge. Pretty-printing for software reengineering. In *Proceedings of ICSM 2002*, pages 550–559. IEEE Computer Society Press, Oct. 2002.

[5] M. de Jonge. Pretty-printing for software reengineering. In *ICSM 2002, 3-6 October 2002, Montreal, Quebec, Canada*, pages 550–559, 2002.

[6] I. Kurtev, J. Bézivin, and M. Aksit. Technological spaces: An initial appraisal. In *International Symposium on Distributed Objects and Applications, DOA 2002*, 2002.

[7] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10: 707, Feb. 1966.

[8] S. McConnel. *Code Complete*. Microsoft Press, 1993.

[9] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman. Program indentation and comprehensibility. *ACM*, 26 (11):861–867, 1983.

[10] D. C. Oppen. Prettyprinting. *ACM TOPLAS*, 2(4):465–483, 1980. ISSN 0164-0925.

[11] T. Parr, F. Zhang, and J. Vinju. Codebuff, June 2016. URL `https://github.com/antlr/codebuff/tree/1.5.1`.

[12] M. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Trans. Softw. Eng. Methodol.*, 5 (1):1–41, Jan. 1996.

[13] M. G. van den Brand, A. T. Kooiker, J. J. Vinju, and N. P. Veerman. A language independent framework for context-sensitive formatting. In *CSMR 2006*, pages 10–pp. IEEE, 2006.

[14] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Trans. Softw. Eng. Methodol.*, 5(1):1–41, 1996. ISSN 1049-331X. .

[15] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *CC '01*, volume 2027 of *LNCS*. Springer-Verlag, 2001.

[16] J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. PhD thesis, U. van Amsterdam, 2005.

[17] D. G. Waddington and B. Yao. High-fidelity c/c++ code transformation. *Electron. Notes Theor. Comput. Sci.*, 141(4):35–56, Dec. 2005. ISSN 1571-0661.

[18] P. Wadler. A prettier printer. In *Journal of Functional Programming*, pages 223–244. Palgrave Macmillan, 1998.