Automatic Code Stylizing

Steven P. Reiss

Department of Computer Science
Brown University
Providence, RI. 02912 USA

spr@cs.brown.edu

ABSTRACT

Coding style is an important aspect of software development. We present a system that uses machine learning to deduce the coding style from a corpus of code and then applies this knowledge to convert arbitrary code to the learned style. We use a broad definition of coding style that includes spacing, indentation, naming, ordering, and equivalent programming constructs. The result provides a more flexible and powerful approach to code stylizing than current techniques.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques — *Pretty printers, program editors.*

General Terms

Languages, Documentation

Keywords

Programming style, formatting, pretty-printing.

1. INTRODUCTION

All programmers have a coding style. The style defines how they like their code to look and how they like their code to read. It defines how they code and how they think about the code they are working on. It affects how they organize their programs and how they work on them.

Programmers are most comfortable working on code in their own style. They find it difficult to work on code in other styles and to force themselves to conform to other programming styles. This is why people often don't like to work on automatically generated code and explains some of the difficulty programmers have working with open source code. Style conflicts also pose problems in team programming projects.

The goal of our project was to convert arbitrary code to an arbitrary style. This can mean converting code to the particular style of a given programmer or converting a programmer's code to the style of a given project. While there are many systems that do this today, they generally require significant work on the part of the programmer to define a relatively large number of parameters and even then they often do not capture the full notion of programming style. We wanted a system that required minimal effort on the part of the programmer and that handled as much of what one would consider coding style as possible.

2. EXAMPLE

Our system is designed so that the programmer provides a sample corpus of code that is formatted in the appropriate style. The system reads and analyzes this corpus, learning the corresponding style. It is then able to apply this style to arbitrary code, converting that code to the appropriate style.

If programmers want to use and modify some open source code, they would have the system read a selection of their code and then format the open source code in their style. Similarly, if programmers wanted to contribute to an open source project, they could develop the code using their own style and then use the system to convert the result to the style used by the open source project. Systems such as user interface builders generate code for use by the programmer. Programmers sometimes have difficulty modifying the resultant code because it is in a foreign style. Our system would let such code be automatically converted to the style designated by the programmer.

As an example of what our approach can do, we took a simple, unformatted file and used the system to format it according to styles learned from different packages. The results are shown in Figure 1. The first part of the figure shows the original, unformatted code. The second part shows it formatted using information from the Azureus open source project (azureus.sourceforge.net). The third shows the file formatted using Eclipse as a base. The fourth shows it formatted according to the openjms open source project (openjms.sourceforge.net). The final one shows it formatted using Sun's j2se model source.

In these examples one can see that the system is able to deal with many aspects of coding style include renaming, alternative programming constructs, breaking lines correctly, indentation, and interline spacing. While the results might not be perfect, they show that the approach can be used to handle a wide variety of styles quite effectively.

3. CODING STYLE

Coding style affects the appearance of the code, how the code is read, how things are found in the code, and how the programming language is used. Our efforts have concentrated

```
package edu.brown.cs.wadi.hump;
import java.text.DateFormat;
import java.util.*;

public class sample { private int the_var; /* cmmt */ public static void main_prog(String[] args) { DateFormat m_format;
    Calendar m_calendar=Calendar.getInstance();
    Random r = new Random(); while (true) { if (r.nextDouble() > 0.5) break; }
    Date date = new Date(); }
}
```

a) The original, unformatted program.

```
package edu.brown.cs.wadi.hump;

import java.text.DateFormat;
import java.util.*;

public class Sample {

  private int theVar;
  /* cmmt */
  public static void mainProg(String[] args) {
        DateFormat mFormat;

        Calendar mCalendar = Calendar.getInstance();

        Random r = new Random();
        while(true) {
          if (r.nextDouble() > 0.5)break;
        }
        Date date = new Date();
      }
}
```

b) Formatted using Azureus (ui)

```
package edu.brown.cs.wadi.hump;

import java.text.DateFormat;
import java.util.*;

public class sample {

        private int theVar;

        /* cmmt */
        public static void
        mainProg(String[] args) {
                DateFormat mFormat;
                Calendar mCalendar = Calendar.getInstance();

                Random r = new Random();
                for (;;) {
                        if (r.nextDouble() > 0.5) break;}

                        Date date = new Date();
                }
}
```

c) Formatted using Eclipse (swt/views)

```
package edu.brown.cs.wadi.hump;

import java.text.DateFormat;
import java.util.*;

public class Sample {

    private int _theVar;
    /* cmmt */
        public static void mainProg(String[] args) {
            DateFormat mFormat;
        Calendar mCalendar = Calendar.getInstance();

        Random r = new Random();
        while (true) {
                if (r.nextDouble() > 0.5)break;
            }
            Date date = new Date();
        }


}
```

d) Formatted using openjms

```
 package edu.brown.cs.wadi.hump;
import java.util.*;
import java.text.DateFormat;

public class Sample {
    private int theVar; /* cmmt */
    public static void mainProg(String[] args) {
        DateFormat mFormat;
        Calendar mCalendar = Calendar.getInstance();
        Random r = new Random();
        while (true) {
            if (r.nextDouble() > 0.5)
                break;
            }
            Date date = new Date();
        }

}
```

e) Formatted using j2se

Figure 1. A sample program formatted using four different styles.

on four aspects of coding style: spacing, naming, ordering, and programming constructs.

The appearance of the code is affected mainly by how white space is incorporated. The principal such use is indentation, where the block level of the code is made readily apparent to the reader. Other uses include how and where potentially long lines are divided, when and how many blank lines are included to increase readability or separate concepts, when spaces are used to separate tokens, and where comments are located relative to the code.

Naming conventions are used to improve code readability, mainly by allowing rapid differentiation between different types of entities, for example constants, fields, methods, and types. In addition, different sets of short names are often used to identify different types of variables, for example *i*, *j*, *k* for integers, *e* for exceptions, or *it* for iterators, with different programmers using different conventions.

Where ordering is not semantic, for example, in defining the fields and methods in a Java file, the order can be considered part of the coding style. Programmers use order to organize their code either to make it easier to find particular fields or methods or to provide a logical order for reading the code. For example, programmers will typically place all the fields of a class either in the beginning or at the end of the class definition.

Most programming languages provide several ways of accomplishing the same task. For example, in Java, one can use an empty *for* statement or a *while(true)* to indicate an infinite loop. The choice of which of these is preferred depends on the programmer and his programming style.

Our approach addresses the above aspects of style. Other aspects could also be considered. For example, the existence of Javadoc comments for different constructs varies according to the coding style. Whether intermediate variables are used in place of complex expressions is a stylistic choice. Similarly, some programmers tend to prefer longer names while other prefer shorter names. Some styles require block comments before each method or routine. While we don't currently handle these, they all could be handled using techniques similar to the ones described in this paper.

4. PRIOR WORK

Formatting code has a long history, although most of the efforts have been on fixing spacing and in particular on handling line breaks and indentation. This has been done based on abstract syntax trees [3,7,12,13,15,17] as well as on purely local information [6,14]. It has been integrated into most programming environments and the subject of stand-alone tools such as *indent*. Theses have been written on understanding what is the best format for program understanding [1]. Other aspects of coding style have received lesser attention.

The state of the art here is exhibited in what is done in modern programming environments such as Eclipse [5]. Eclipse provides a very flexible code formatting scheme, requiring about 124 parameter settings from the user. It also provides support for suffixes and prefixes for some types of names, support for ordering import declarations, and templates for styling newly created code.

This approach has its drawbacks. It is painful to have to worry about defining a large number of parameter settings and attempting to get them correct. Moreover, even with the large number that are there, it is unable to handle all formatting options. For example, we prefer our right braces to be on a separate line that is half indented and Eclipse provides no option for the half indent; similarly, we want spaces around operators in complex expressions, but not in simple ones. Finally, this approach does not handle all aspects of coding style including general orderings, more general naming conventions, and alternative programming constructs.

There has also been significant work on programming (as opposed to coding) style. This work concentrates on how to best use the underlying programming language, pointing out language features that shouldn't be used and various conventions that tend to yield better programs [8,11,16]. Our initial goal was to concentrate on coding style since many of the transformations needed to conform to a programming style are not obvious or transparent to the programmer.

5. A LEARNING-BASED APPROACH

Rather than having the programmer define the style by attempting to anticipate all possible styles and then parameterizing the formatter, we wanted a system that could look at sample code, learn the coding style of that code, and then apply what it learned to reformat other files.

We wanted to address many of the different aspects of coding style. However, the different aspects depend on different properties of the code. Spacing depends on the underlying structure and the local tokens; naming depends on the types of names, how names are used, and specific names for specific data types; ordering depends on the properties of the items being ordered (for example, public versus private) as well as alphabetic comparisons; programming construct alternatives depend on structure matching. Moreover, ordering is constrained in some contexts by usage in languages like C where names must be declared before they are used; and naming conventions may be constrained by external conventions, such as when a Java class implements an interface from some other library the names of the overridden methods are otherwise constrained.

To accommodate these differences, we broke the task into phases, with each phase handling learning and applying what was learned to a particular aspect of coding style. The initial set of phases includes dealing with spacing, handling naming conventions, handling short names, handling orderings, and handling programming constructs. For all but the last of these we determined an appropriate feature set that defines the context for formatting decisions and an appropriate output set which defines the possible results of the decisions. We then experimented with different learning algorithms and feature set variations to determine what worked best for this particular component. Finally, we used appropriate techniques to reformat a resultant file based on the learning results.

6. FORMATTING

Spacing involves determining how much and what type of white space to put between successive tokens in the program. This implies decisions on when and where to split lines, how much to indent, and where to place comments on a line.

Our approach to handling formatting is to determine for each token what type of white space should precede that token. To do this by learning we defined a set of sixty features that characterize a token. These features capture the information that might be relevant to spacing. They include the length and type of the current token; whether the current token is a single line comment; if the token was preceded by a blank line; information about the previous token including where it occurred on the line, its type, and its length; information about the three preceding tokens including their type and, if they occurred on the same line as the prior token, their spacing and column positions; structure information derived from the abstract syntax tree including the node type for this token, the node types for the parent and grandparent of the token, the depth of expressions and blocks; the types of the first three tokens on the current line (assuming this token is part of the same line) and the first three tokens on prior line; the starting column of the last line and the last non-comment line; the number of immediately prior comment tokens; the current level of nesting of parenthesis, braces, and brackets; the column location of the last open parenthesis, bracket, and brace; the prefix on the last and current import; and the brace level at the start of the last switch statement.

Most of these correspond to what the programmer uses intuitively to determine spacing. The features based on the abstract syntax tree are designed to capture the local structural context, while those that characterize prior tokens in various ways attempt to represent the current formatting context. Others are more specialized and were added to avoid specific problems that would otherwise arise, typically because of semantic information. For example, the notion of the prior import attempts to capture related import blocks which programmers often space between, while the notion of blank line preceding this line in the original file attempts to capture the fact that the programmer thought this line might represent a new thought and might need to be spaced accordingly.

From this set of features we wanted to learn what spacing to put before the token. We considered several alternative approaches here. The simplest approach was to learn the number of blank lines and the number of spaces after those lines and before the token. Here a single space would be represented as 0 lines and 1 space, while a blank line followed by an indent of 9 space would be represented as 2 lines and 9 spaces.

While this approach is simple, it does not capture the programmer's intuition of how spacing is done. To see if a better model would result in better results, we considered several alternatives. First we let the result reflect indentation information by indicating lines with a plus or minus indent in addition to lines that set a fixed indent for later use. We denote this as I=I. Second, we did the same, but we ignored the indentation of comment lines which are often skewed differently (I=C). In parallel with these alternatives, we also considered a mode based on column of tabs. Here the output could indicate a specific column (S=C); a change in the column position from the previous line (S=CL); a specific number of tabs (S=T); or a change in the number of tabs (S=TL). This gave us fifteen different alternatives, 3 ways of handling indentation (none, with and without comments), cross 5 ways of handling columns.

Finally, we needed to interpret the output from the learner. Typically the learning method yields a set of probabilities for the different alternatives. The obvious choice is to just take whatever alternative has the highest probability and use it. While this works fairly well, there are cases where several alternatives have roughly equal probabilities. In this case, we looked at the spacing that was originally present in the program and, if this was represented by one of the likely outputs, we maintained that spacing. How much we rely on the original spacing is dependent on a confidence level setting, with a confidence level of one indicating that we ignore it and higher confidence levels relying on it more.

We ran experiments that tested a variety of different learning algorithms with these fifteen possible outcomes to determine which learning algorithm worked best and which approach to defining spacing gives the most accurate results. These experiments were run by reading a corpus of about 26,000 lines of source written by a single author. The style of this code was fairly consistent but not completely so as the code was written over a period of years and style was not a primary consideration. The spacing learned was then applied to 25 files (about 12,000 lines of source and separate from the corpus) to determine performance and effectiveness. These files were preformatted by the same author as the original code corpus, so that in theory there should be few if any style changes required. Where changes were indicated, we assumed that the learning methods were in error. Finally, one of these files in particular was chosen for a manual evaluation because it was reasonably large and exhibited a diversity of different styling situations. This was used to determine the effectiveness of the error metrics.

The results of these experiments are sampled in the various tables. The first, Figure 2, shows the best result for each of the learning methods. The output column denotes the output technique that was used using the codes noted previously.

We used learning methods from three different packages. From the Weka system [9,18] we used the J48 classifier (a C4.5 decision tree learner), the SMO classifier (a support vector machine), the K* instance-based learner, and JRIP (a version of the RIPPER rule learner). For each of these we tried it with numeric or enumeration values and with all boolean values. From the Mallet system [10] we used a naive Bayes learner, a Maximum Entropy learner, and a C4.5 tree learner. We also tried the maximum entropy learner from the OpenNLP project [2]. Other methods were tried but they were not able to complete the learning process using less than 24G of memory or 1 week of CPU time.

For each method we report the accuracy in two ways. The first is the raw percentage of spacing decisions that differed from the "correct" file. Here any difference in white space was considered an error. While this provides a good indication of how well the method did, not all spacing decisions are equal. For example, bad indentation decisions can be very annoying to the programmer, while the difference between 4 and 5 blank lines between methods is insignificant. To account for these differences we analyzed the types of differences that were detected and created a score that provides different weights to the different decisions. For example, a bad indentation decision has a weight of 8 for a token and 2 for a comment while the difference in line spacing when the space is over 3 is only penalized by 0.25. The result is reported in the score column.

The remaining columns show the size of the model generated by the learner and the CPU time in seconds taken to do the learning and to do the formatting of the 25 files. The model size is the size of the data file that is stored by the learning method. Because different methods use different storage formats (e.g. text, binary, compressed), these numbers should be taken as approximate.

Method	Output	% Difference	Score	Model Size	Learning Time	Format Time
Weka J48 Tree	I=I, S=T	3.0%	2.8%	348.7M	439.5	60.2
Weka J48 Tree Bool	I=I, S=T	3.5%	3.5%	29.3M	37993.2	699.9
Weka SMO	I=I, S=T	0.0%	0.0%	2412.7M	95861.8	9051.4
Weka SMO Bool	I=I, S=C	0.0%	0.0%	2475.2M	0.0	38688.5
Weka K-Star	I=C, S=TL	6.0%	9.0%	52.4M	546024.1	370374.5
Weka JRip Rules	I=I	3.6%	4.0%	807.9M	36235.5	80.1
Mallet Naive Bayes	I=C, S=TL	6.9%	9.1%	5.3M	1484.7	348.5
Mallet Max Ent	I=I, S=T	2.1%	2.0%	4.7M	14800.0	55.6
Mallet Max Ent Bool	I=I, S=T	2.0%	1.9%	4.8M	9002.6	109.9
Mallet C45 Tree	S=C	3.8%	3.1%	453.0M	88806.6	45.0
Mallet C45 Tree Bool	S=C	4.7%	3.5%	505.7M	125683.2	65.6
OpenNlp Max Ent	I=C, S=TL	8.1%	8.6%	1.1M	679.6	24.9

Figure 2. Results for different learning methods for learning code spacing.

Method	Output	% Difference	Score	Model Size	Learning Time	Format Time
Mallet Max Ent	I=I, S=T	2.1%	2.0%	4.7M	14800.0	55.6
Mallet Max Ent	I=C, S=TL	2.1%	2.0%	5.4M	10085.0	102.1
Mallet Max Ent	I=C, S=T	2.1%	2.1%	4.9M	10896.6	75.5
Mallet Max Ent	I=I, S=TL	2.1%	2.1%	5.2M	11648.3	86.6
Mallet Max Ent	I=I	2.1%	2.1%	5.3M	15580.9	74.2
Mallet Max Ent	I=C	2.1%	2.1%	5.4M	11368.7	69.4
Mallet Max Ent	S=C	2.4%	2.2%	5.0M	6924.4	66.5
Mallet Max Ent	S=CL	2.4%	2.3%	7.7M	18353.2	148.8
Mallet Max Ent	I=I, S=CL	2.1%	2.6%	7.0M	20579.1	72.0
Mallet Max Ent	I=C, S=C	2.2%	2.6%	5.4M	19705.7	88.3
Mallet Max Ent	I=C, S=CL	2.1%	2.6%	7.1M	14572.4	70.5
Mallet Max Ent	I=I, S=C	2.1%	2.8%	5.0M	9443.6	65.6
Mallet Max Ent	S=T	2.4%	4.1%	4.7M	10151.1	61.4
Mallet Max Ent		2.4%	4.2%	5.3M	10908.8	85.1
Mallet Max Ent	S=TL	2.4%	4.2%	5.3M	10370.2	64.6

Figure 3. Differences based on output method.

The second table, Figure 3, shows the different alternative outcomes for the Mallet maximum entropy learning method which did well on the overall approach. This shows how the use of the current indentation information (the various I= trials) significantly improves the result, while the effect of recording tab and column positions is minor.

The third table, shown in Figure 4, shows the effect of varying the confidence level for both the Mallet MaxEnt and the Weka J48 Tree learning methods. The confidence level here varies from 1 (ignore the current source) to 5 (if the probability of the current spacing is within a factor of 5 of optimal, use the current value). As expected, we get better scores as the confidence level increases. There are two things to note here. First, even with a low confidence level, both these methods did quite well. Second, the confidence level has more effect on learning methods that assign some probabilities to all scores such as MaxEnt than it does on learning methods that artificially restrict the set of outcomes using mechanisms such as a decision tree.

From these results one can see it is possible to use various algorithms to learn the complete program spacing style. Several algorithms can be used effectively. Some, such as the SMO classifiers, while very accurate, are probably too expensive to be practical. Of the remainder, the Mallet maximum entropy learner and the Weka J48 rule-based learner seem to offer the best compromise between accuracy and performance.

7. NAMING

The second aspect of style we addressed was naming. We did this in two steps, first attempting to learn the naming conventions used in the code and then attempting to learn if there were standard short names for specific variables. Our approach for both of these was to choose a set of features that might influence naming, choose an appropriate output representation and then to apply different learning algorithms to see what worked and what worked best.

Method	Output	Confidence	% Difference	Score
Weka J48 Tree	I=I, S=T	1.0	3.1%	3.0%
Weka J48 Tree	I=I, S=T	2.0	2.9%	2.8%
Weka J48 Tree	I=I, S=T	3.0	2.7%	2.7%
Weka J48 Tree	I=I, S=T	4.0	2.6%	2.6%
Weka J48 Tree	I=I, S=T	5.0	2.6%	2.5%

Method	Output	Confidence	% Difference	Score
Mallet Max Ent	I=I, S=T	1.0	3.1%	3.2%
Mallet Max Ent	I=I, S=T	2.0	2.5%	2.4%
Mallet Max Ent	I=I, S=T	3.0	2.1%	2.0%
Mallet Max Ent	I=I, S=T	4.0	2.0%	1.8%
Mallet Max Ent	I=I, S=T	5.0	1.8%	1.7%

Figure 4. The difference the confidence value makes for two learning methods.

The twenty-four features that we used describe both the properties of the name and its context. The features describing the name's properties include the modifiers used in defining the name (static, abstract, final, native, transient, synchronized, volatile); the protection of the name (public, private, protected, package protected); the kind of object being named (interface, class, method, block, for variable, catch variable, annotation, package, enumeration constant); if the name represents a method, whether it is a getter or setter method; and if the name represents a variable, its data type. While it is impractical to represent all possible data types here, we do identify primitive types, Java standard types such as String and Object, and types that inherit from Error and Exception. We also identify whether the name is short (3 or fewer characters) and whether it is a special name required by the underlying programming language such as *serialVersionUID*.

The context of the name is provided by a separate set of features. These include the kind of scope the name is defined in (package, interface, class, method, for, catch, block, enumeration), the protection level of the enclosing object, the degree of nesting of loops, blocks, and classes or interfaces, and whether the name is actually used somewhere in the code or if it is just defined.

Representing naming conventions is a bit tricky. We attempt to encode most of the different naming strategies that have been proposed using a set of seven properties:

1. Whether the name is all upper case, all lower case, or mixed case (new words start with an upper case letter).
2. Whether the first character is upper or lower case.
3. Whether words are separated with underscores or not.
4. The number of initial underscores (0-2).
5. The number of final underscores (0-2).
6. The number of underscores separating the first word from the remainder (0-3).
7. Whether the name starts with a standard prefix (get, set, is, the, f, my) or with the name of the current package or class.

Splitting names into words is approximated by using a dictionary and by looking at clues from the code being analyzed (such as the words or word fragments that occur in names that are known to be split into words).

To identify short names, we use the same set of features. Here the output representation is the set of short names (again 3 or fewer characters) that appear in the sample corpus along with a special symbol indicating that the name was not short.

Again, we took into account the probabilities when looking at the result of name learning, using the original name if the learning method says that it was a probable choice based on the confidence level. This tended to occur far more often with naming than with spacing because some aspects of the naming style were not used consistently in the corpus. For example, in the code that was being analyzed, *get* was not used consistently for get methods, nor were package or type prefixes, nor were short names in many circumstances.

The results from the different learning methods are shown in Figure 5. There were about 2400 distinct name declarations that were considered as potentials for change. The table shows the results both for name style and for short names. The first column indicates the learning method used; these are the same as were used in learning spacing. The second column provides the number of names where the computed style differed from the actual style in the test file. The third and fourth columns provide the size of the saved learning model and the time in seconds required to compute it respectively. The remaining columns show the differences for suggested short names and the associated model size and time.

Most of the methods do quite well on computing the naming style, choosing a naming style for each name that matches the "correct" one. Most differences here were due to names inherited from outside classes. Most of the methods also detected no changes in short names. This was due mainly to the fact that the corpus is not completely consistent in its use of short names and the confidence level pushed ambiguous names to their previous values.

8. ORDERING

The next part of coding style that we handle involves orderings where the orderings do not have a semantic significance. In particular, we consider import declarations, class definitions in a compilation unit, declarations (fields, methods, and nested types) in a class, interface or enumeration definition, and modifiers for type, field and method definitions. We also note

Method	Style Diff	Style Size	Style Time	Short Diff	Short Size	Short Time
Weka J48 Tree	0.3%	12.2M	28.3	0.0%	1.3M	5.9
Weka J48 Tree Bool	0.2%	6.1M	70.7	0.0%	1.1M	13.7
Weka SMO	0.0%	490.4M	477.4	0.0%	71.6M	2343.7
Weka SMO Bool	0.0%	490.2M	761.9	0.0%	71.4M	2580.2
Weka K-Star	0.2%	3.2M	1861.1	0.1%	3.1M	1859.3
Weka JRip Rules	1.0%	9.2M	52.8	0.0%	54.1M	157.1
Mallet Naive Bayes	4.8%	16.7K	111.0	0.0%	270.8K	53.0
Mallet Max Ent	0.2%	15.9K	12.5	0.0%	261.6K	68.8
Mallet Max Ent Bool	0.2%	16.1K	12.3	0.0%	261.8K	120.4
Mallet C45 Tree	0.2%	204.3K	17.9	0.0%	1.3M	7.5
Mallet C45 Tree Bool	0.1%	273.4K	19.6	0.0%	1.3M	9.2
OpenNlp Max Ent	0.1%	32.0K	4.9	0.0%	573.1K	34.2

Figure 5. Result of learning naming conventions and short names.

Method	Check Diff	Sequence Diff	Diff Length	Ulam Average	Ulam Diff	Size	Time
Weka J48 Tree	4.5%	2.6%	29.2	0.1	3.3	83.4K	110.5
Weka J48 Tree Bool	4.5%	2.1%	26.4	0.1	3.6	74.1K	242.5
Weka SMO	26.5%	15.9%	8.7	0.7	4.2	44.0M	9993.6
Weka SMO Bool	26.5%	15.9%	8.7	0.7	4.2	43.0M	9676.8
Weka K-Star	2.5%	2.3%	23.1	0.1	2.8	38.9M	294861.1
Weka JRip Rules	11.4%	1.3%	28.0	0.1	3.9	59.8M	691.6
Mallet Naive Bayes	3.0%	3.5%	20.6	0.1	3.2	4.9K	67.6
Mallet Max Ent	3.2%	9.0%	8.0	0.1	1.5	4.5K	64.8
Mallet Max Ent Bool	3.1%	9.0%	8.0	0.1	1.5	4.6K	39.2
Mallet C45 Tree	12.4%	1.9%	23.1	0.1	4.1	2.7M	77.1
Mallet C45 Tree Bool	13.9%	2.2%	25.3	0.1	3.2	2.7M	80.3
OpenNlp Max Ent	1.4%	1.0%	36.7	0.0	3.7	7.8K	17.3

Figure 6. Result of learning orderings of declarations and modifiers.

that some semantic reorderings such as reordering parameters for a method or declarations at the start of a block should also be possible using Eclipse's refactoring capabilities, but we have not attempted to do so.

There are several approaches that can be taken to learning an ordering. Our approach is to learn, for each potential pair of elements in the order, which is likely to come first. This is done by providing the learning method with all $n^2/2$ pairs of elements for each n element sequence along with the original ordering. To determine how to reorder elements when we want to apply a format, we look at all $n^2/2$ pairings of elements that can be reordered. For each, we use what was learned to determine which should come first, using the given order if the answer is not clear. We then add a score of -1 to the first and a score of +1 to the second. Then we order the actual elements by their total scores and assume that is the final ordering.

We again determined a set of features relevant for a particular coding style to the ordering of modifiers and definitions. This includes properties of each of the definitions and comparisons between the two. The forty-one properties we consider include modifiers, annotations, protection, whether a field is initialized, whether a method is a getter or a setter, whether an import is for a specific element or a whole package, and, the name of the first part of an import.

For comparisons between two elements, we first note if one precedes the other alphabetically both where case is considered and where case is ignored. Next we note if either of the definitions makes use of the other, i.e. if both are methods, if the one calls the other, or if one is a field and the other is a method, if the method accesses the field (or the field calls the method in its initializer). Finally, if both are enumeration constants, we note which has a lower associated value.

We again ran each of the learning methods for the sample corpus and analyzed the result using our test corpus. This resulted in about 25,000 different pairwise comparisons for about 720 different sequences with an average length of 2.5.

The results are shown in Figure 6. For each learning method, the first column reports where there were differences in determining which of a pair of elements comes before the other. The second column reports the percent of actual orderings that differed after all comparisons of elements in that ordering were taken into account. The next column reports the average length of orderings that were different.

```
void STATEMENT_forever() {
  switch (0) {
    case 1 :
      for ( ; ; ) HUMP_STATEMENT_1();
    case 2 :
      while (true) HUMP_STATEMENT_1();
  }
}

void STATEMENT_forblock() {
  for ( HUMP_EXPRESSION_1(); HUMP_BNULLEXPRESSION_2(); HUMP_EXPRESSION_3()) HUMP_NONBLOCK_4();
  for ( HUMP_EXPRESSION_1(); HUMP_BNULLEXPRESSION_2(); HUMP_EXPRESSION_3()) { HUMP_NONBLOCK_4(); }
}

void STATEMENT_whileblock() {
  while (HUMP_BEXPRESSION_2()) HUMP_NONBLOCK_4();
  while (HUMP_BEXPRESSION_2()) { HUMP_NONBLOCK_4(); }
}

void STATEMENT_increment() {
  HUMP_IVARIABLE_1 += 1;
  HUMP_IVARIABLE_1++;
  ++HUMP_IVARIABLE_1;
}
```

Figure 7. Example equivalent construct definitions.

Node Type	# Matches	# Different
ExpressionStatement	50	1
ForStatement	52	0
MethodInvocation	14	0
WhileStatement	5	0

Figure 8. Results of finding common programming constructs.

Just looking at differences here does not truly reflect how well the methods do. Instead, one wants to consider how different the outcome is to the original. To do this we compared the output permutation with the input permutation using Ulam's distance metric [4]. This metric effectively provides the edit distance between permutations and is a more accurate means for comparison than the number of items that retain their order. The next two columns show the average Ulam distance over all cases and over those cases that actually differ. The final two columns provide the size of the saved learning model and the time in seconds to compute that model.

This table shows there are considerable differences among the learning methods. This is probably due to the fact that the corpus being used is not as consistent in terms of ordering as it is in terms of naming or spacing and the sensitivity of the various methods to exceptions differs. Overall, the Weka J48 tree learner tended to do quite well; the Mallet MaxEnt learner, while it had a low difference in terms of checks, actually resulted in more sequences that differed from the original program. The small values for the Ulam difference compared to the large average length of sequences that differed indicates that the number of actual changes is quite small. Overall, the results show that the algorithms can effectively learn ordering styles.

9. PROGRAM CONSTRUCTS

The final aspect of coding style that we handle involves equivalent programming constructs. Programming languages often have multiple ways of saying the same thing, and the choice of which alternative is preferable is often considered part of ones coding style.

Rather than attempt to find such constructs automatically, we created a file that let the user define any number of equivalent patterns and then had the system check for these. The file is a Java class with one method for each alternative. The contents of each method contain two or more equivalent statements or expressions, with special variable names being used to denote arbitrary code elements of various types.

Examples from the file are shown in Figure 7. The first shows the equivalence between two ways of expressing an infinite loop. The second states a preference of whether a for loop with a single statement should have that statement appear in a block or not. The third does the same for while loops. The fourth illustrates several statements that increment a variable. (Note that these are defined as statements and not as expression components.)

Our approach to learning programming constructs is simpler than for the other style components. Here we first translate each of the given patterns into an abstract syntax tree pattern. Then we look through the sample corpus and count all instances of each of the patterns in the corresponding code. We then look at the count differentials between the different instances of each of the equivalent patterns and use this as the learning model.

The model size here is insignificant (under 1K), as is the time to accumulate the pattern information. Figure 8 shows the result of applying what was learned from the sample corpus to the test corpus. As would be expected, there were few changes required in the sample code, with the only difference being an instance of an expression statement that was converted from *i++* to *++i*.

Many other equivalent constructs could have been included by extending the framework. For example, one style choice is whether fields defined in an interface are explicitly declared with static, final, and public. This would require that the patterns be limited to a particular context which would require an extension to the pattern language. Other choices, for example when one compares a variable to a constant string which is placed inside the Java call to *equals*, would require semantic checks to ensure that the variable represented by the string is not null. Other possibilities such as when generics are used in Java might require whole program analysis to determine the appropriate types.

10. APPLYING THE LEARNED STYLE

While each of the aspects of style can be learned independently, applying them to a new source file is a bit more complex. There are two inherent problems here. The first is that the different aspects of code style are dependent on one another. For example, spacing can depend on the length of names and replacement programming constructs while sequencing can depend on the exact names. The second problem is that the features for spacing are computed for each token are based on the computed spacing for previous tokens, information that will change when the code is reformatted.

To handle all the different types of style changes we use the Eclipse environment. Here we first create the abstract syntax tree for the file being reformatted. Then we edit the tree to take into account naming, ordering and programming constructs. Finally, we output the tree using spacing information.

The first step in reformatting is to handle name changes. Here we first compute all potential name changes. Next we use Eclipse to detect if a potential name change is possible. Names that are declared, for example, in interfaces outside the project or the code being restyled, should not be changed since doing so will result in an incorrect program. After we have determined that a name can be safely changed, we then use Eclipse's refactoring capability to effect the change of the definition and all uses of the given name.

Next we compute all ordering changes and modify the Eclipse abstract syntax tree accordingly. After this we search the abstract syntax tree for programming constructs.When we find one, we check if other alternatives have a significantly higher chance of occurring and, if so, we replace the original code with the more prevalent equivalent.

The final step is to use spacing information to generate the resultant program. We first convert the abstract syntax tree to text and then convert the text to tokens. We process the resultant token sequence as the input for spacing. At each point, we compute the various features for spacing, using information from the token and the abstract syntax tree. We also maintain the position information for the previous tokens and the previous lines so that the feature set reflects the code as it is being formatted. The resultant feature set is passed to the learning method for spacing to determine the resultant spacing for the given token, this spacing and the token itself are output, and the internal model is updated for the next token.

11. CONCLUSIONS

This paper and the various experiments we have conducted demonstrate that it is both possible and practical to learn many aspects of coding style from a corpus and then to apply that knowledge to reformat arbitrary code in the corresponding style. This approach is much simpler and more comprehensive than current approaches based on large numbers of user-settable parameters.

The example shown in Section 2 demonstrates that the system is capable both of completely ignoring the original formatting information and of adapting itself to a variety of different styles.

The various experiments described in the paper demonstrate that different learning methods can be used effectively and that the corpus of code representing the target style does not have to be that large or completely consistent. Indeed, one of the reasons we restricted the corpus size to about 26,000 lines of source was that the time spent in the different learning algorithms, which is already quite large, increased with the corpus size. We found that accuracy generally increased with the size of the corpus both because we achieved better coverage of fringe cases and because style differences in the corpus became less significant. However, the increase in accuracy was not significant beyond 20,000-30,000 lines of source.

There are several ways that this work can be extended. First, the notion of coding style can be broadened in the various ways that were described earlier, both in terms of overall style and in terms of alternative programming constructs. Second, the tool, which currently runs in stand-alone mode, could be integrated into a programming environment such as Eclipse. Third, while integrating the tool into the programming environment, it should be possible to have the programmer identify incorrect style transforms and to have the system learn from these.

Finally, we note that the source code for this effort is available as the hump package in the wadi system, which can be found at ftp://ftp.cs.brown.edu/u/spr/wadi.tar.gz.

12. ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation through grant CCR0613162. We also thank the various referees for their helpful comments.

13. REFERENCES

1. Ronald M. Baecker and Aaron Marcus, *Human Factors and Typography for More Readable Programs*, Addison-Wesley (1990).
2. Jason Baldridge, Tom Morton, and Gann Bierner, "OpenNLP: The Maximum Entropy Framework," *http://maxent.sourceforge.net/about.html*, (October 2001).
3. Robert D. Cameron, "An abstract pretty printer," *IEEE Software* Vol. **5**(6) pp. 61-67 (November 1988).
4. D. E. Critchlow, *Metric Methods for Analyzing Partially Ranked Data*, Springer-Verlag, Lecture Notes in Statistics, Volume 34 (1985).
5. Erich Gamma and Kent Beck, *Contributing to Eclipse*: *Principles*, *Patterns*, *and Plug-ins*, Addison-Wesley (2004).

6. James Gosling, *Unix Emacs*, Carnegie-Mellon Computer Science Department (August 1982).
7. Matti O. Jokinen, “A language-independent prettyprinter,” *Software Practice and Experience* Vol. **19**(9) pp. 839-856 (September 1989).
8. Brian W. Kernighan and P. J. Plauger, *The Elements of Programming Style*, McGraw-Hill (1974).
9. Richard Kirkby and Eibe Frank, “WEKA explorer user guide for version 3-5-3,” *University of Waikato*, (June 2006).
10. Andrew Kachites McCallum, “MALLET: a machine learning for language toolkit,” *University of Massachusetts Computer Science*, *http://mallet.cs.umass.edu*, (2002).
11. Scott Meyers, *Effective C++*, Addison-Wesley (1997).
12. Dereck C. Oppen, “Prettyprinting,” *ACM Trans. Programming Lanaguages and Systems* Vol. **2**(4) pp. 465-483 (October 1980).
13. Steven P. Reiss, “PECAN: program development systems that support multiple views,” *IEEE Trans. Soft. Eng.* Vol. **SE-11** pp. 276-284 (March 1985).
14. Steven P. Reiss, “The Desert environment,” *ACM TOSEM* Vol. **8**(4) pp. 297-342 (October 1999).
15. Lisa F. Rubin, “Syntax-directed pretty printing: a first step towards a syntax-directed editor,” *COMPSAC 1981*, (1981).
16. Allan Vermeulen, Scott W. Ambler, Greg Bumgardner, Eldon Metz, Trevor Misfeldt, Jim Shur, and Patrick Thompson, *The Elements of Java Style*, Cambridge University Press (2000).
17. Philip Wadler, “A prettier printer,” *Bell Laboratories*, (March 1998).
18. Ian H. Witten and Eibe Frank, *Data Mining*: *Practical Machine Learning Tools and Techniques*, *2nd Edition*, Morgan Kaufmann (2005).