DUETCS: Code Style Transfer through Generation and Retrieval

Binger Chen
TU Berlin
Berlin, Germany
chen@tu-berlin.de

Ziawasch Abedjan
Leibniz Universität Hannover & L3S Research Center
Hannover, Germany
abedjan@dbs.uni-hannover.de

2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) | 978-1-6654-5701-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE48619.2023.00198

***Abstract*—Coding style has direct impact on code comprehension. Automatically transferring code style to user's preference or consistency can facilitate project cooperation and maintenance, as well as maximize the value of open-source code. Existing work on automating code stylization is either limited to code formatting or requires human supervision in pre-defining style checking and transformation rules. In this paper, we present unsupervised methods to assist automatic code style transfer for arbitrary code styles. The main idea is to leverage Big Code database to learn style and content embedding separately to generate or retrieve a piece of code with the same functionality and the desired target style. We carefully encode style and content features, so that a style embedding can be learned from arbitrary code. We explored the capabilities of novel attention-based style generation models and meta-learning and implemented our ideas in DUETCS. We complement the learning-based approach with a retrieval mode, which uses the same embeddings to directly search for the desired piece of code in Big Code. Our experiments show that DUETCS captures more style aspects than existing baselines.**

I. INTRODUCTION

Code style refers to a set of rules on how code should be organized and includes aspects, such as formatting, naming, ordering of code blocks, comment usage, code constructs, and modularization. Code style generally does not interfere with the code semantics and executability, but it has a direct impact on code comprehension. For example, the formatting style highlights the control flow [18], naming style reflects the connection between code and its problem domain [20], [32]. In general, the readability of code plays an important role in software development [24], [6], [31] and code maintenance [5], [31]. Developers spend about 80% of their time maintaining code, half of which is spent on code comprehension [7]. The Apple incident of SSL/TLS certificates shows the impact of code style on codebase's reliability [4], [3].

Different programmers have their own writing styles. When multiple programmers work on the same projects and there is fluctuation in developer teams, different code styles in code artifacts becomes a practical hurdle in code comprehension. Unifying the different styles can facilitate cooperation and follow-up maintenance. Code comprehension becomes a more general problem when considering the large amounts of open-source code. A study shows that almost half of the Java snippets on Stack Overflow are not self-explanatory [33]. To improve code comprehension in open-source projects, repository maintainers have to keep the repository style consistent. This effort can be tedious when a repository receives many pull requests from different contributors.

On a different note, with the rising demand for automation in coding, machine-generated code is becoming ubiquitous [17], [29], [27]. Maintenance of code style and automated adaptation of style in generated code becomes more and more important. In this paper, we explore the potential of Big Code, and leverage learning-based feature representing methods to assist programmers in code stylization that works for arbitrary code styles and programming languages.

State of the art. To the best of our knowledge, there is no existing work that performs full stylization on *an arbitrary piece of code*. The most common methods are rule-based linters, formatters, which are limited to a few pre-defined style rules. There has also been substantial work on using machine learning to learn style features from code [8], [26]. However, these approaches are limited to formatting issues and do not capture style aspects concerning naming and structuring. Moreover, they usually require human intervention to set the formats or parameters. The most recent work STYLE-ANALYZER is fully unsupervised [23] but limited to extracting formatting rules.

Our research question is whether it is possible to automatically extract stylization signals and enforce them on a piece of code. In particular, we have to distinguish style and content elements so that we can obtain code that retains its original functionality but is in the desired *target style*. For this purpose, we explore techniques that benefit from large repositories of code and employ novel representation learning techniques to separately learn style and content features. Developing such an approach requires us to overcome several technical **challenges**:

- There is no labeled dataset for code style transfer. It is not upfront clear how style transfer can be automatically learned and assessed.
- Style transfer comprises multiple tasks, such as detecting the style differences, analyzing the code semantics, and generating code. Integrating these steps into an end-to-end process requires a generalizable content and style representation. We have to distinguish content and style features in raw code and then encode them in a unified feature representation.

To this end, we study self-learning on Big Code and present **DUETCS** that implements a novel code feature representation

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00198

without any pre-defined stylization rules to assist the **c**ode **s**tyle transfer task of transferring a piece of code to an arbitrary style. To achieve this, we separate style features from content features. To capture comprehensive style features, DUETCS generates a style embedding by encoding text style and structure style features. To capture the content features, we adapt code representation previously used for cross-language code retrieval by stripping it from style information [16], [15]. To train the feature embedding, we use a meta-learning-based network that self-trains using the Big Code and its meta-data. This way, we avoid human labeling efforts. We further explore how our representation can be used in a *a retrieval mode* that provides the option to directly search for the potential result in the Big Code database. In this paper, we analyze the advantages and disadvantages of these two strategies. In a practical setting, DUETCS simultaneously runs both generation and retrieval modes and evaluates their results to return the best possible result to the user. In short, our **main contributions** are:

- We explore how two different self-learning modes - retrieval and generation - based on Big Code, can assist automatic code style transfer without a labeled dataset.
- We propose a novel code feature learning approach based on meta-learning that can separate style features from content features. We construct a style feature representation that captures both text and structure style. We design a novel style generation model that encodes the source code and pays more attention to the style features.
- We propose an LSTM-based model that directly generates the raw transferred code by combining the style feature and content feature embeddings.
- We conduct extensive experiments that show the strengths and weaknesses of an unsupervised stylization approach. We discuss the advantages and disadvantages of the two modes, learning and retrieval. Our results show that with specific safety mechanisms the proposed approaches generate useful stilization recommendations.

II. RELATED WORK

Our work is related to automated code stylization and program feature representation.

Code style transfer. There are rule-based tools, such as code linters and formatters, which validate the compliance of a codebase with a language's style guide. They can be included in IDEs, such as INTELLIJ or Eclipse. There are also standalone tools, such as pylint, ESLINT, and Gnu Indent. Those manually created rules are usually language specific and too general to capture personalized style. There are learning-based approaches that learn common coding styles observed in codebases. They convert code into a representation, such as one-hot token embeddings or parse trees. Then they learn formatting conventions and generate either explicit [8], [23] or abstract [26] rules from them. For example, NATURALIZE [8] is a language-agnostic code formatting suggester, which uses statistical natural language processing to learn the style of a codebase and make revision suggestions. CODEBUFF [26] uses

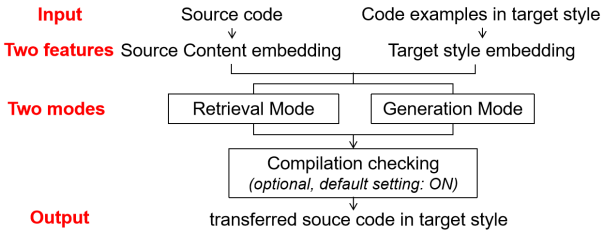

Fig. 1: DUETCS Overview

the k-Nearest Neighbor model to predict style elements, such as newlines and spaces. These methods require the user to pre-define the formatting style or to set configurations. Recent work STYLE-ANALYZER [23] is an unsupervised formatting tool. It uses a language model to learn the underlying formatting style of a repository, then trains a decision tree forest model to extract the formatting rules. STYLE-ANALYZER and CODEBUFF only address formatting style. These approaches focus on single token prediction, which does not address style problems involving multiple tokens or the code structure. In contrast, DUETCS automatically learns abstract style patterns from code examples in target style and considers more comprehensive code style aspects and languages.

Program representation. Constructing a program feature representation is an essential task for many applications. Allamanis et al. propose a representation that treats a program as a sequence of text tokens to summarize the code and predict method names [10]. This approach is oblivious to code specific features, such as program syntax and token types. Their follow-up work includes data flow and type hierarchies in the representation [9]. But it is language-specific. Alon et al. propose a more general representation that uses linear paths in the abstract syntax trees (AST) as features to predict program properties [12]. To obtain task-independent representations, they further propose Code2vec, which leverages a tree-based neural network to encode these paths and generate more abstract representations [13]. However, their model cannot capture unseen data. To learn unseen paths, Code2seq represents the syntactic paths node-by-node using LSTMs [11]. Another representation is based on the natural language model Transformers [34], which uses self-attention to encode context information. For example, CODE TRANSFORMER combines distances computed on structure and context in the self-attention operation [36]. All these representations capture general program features. However, to only transfer the code style, we need to separate the content features from the style features. Finally, approaches for cross-language program retrieval [16], [15] use AST-based representations. Again, these approaches do not aim to separate the style and content features. While we reuse some of their ideas we develop representations that are tailored to stilization.

III. OVERVIEW

The workflow of DUETCS is shown in Figure 1. The user inputs a program P_{in} whose style is intended to be transferred. We denote its content as C_{in}. Another input from

the user is a set of code examples $\{P_1^S, P_2^S, ...\}$ written in the desired target style S. Our system provides a large program repository or database D that is originally crawled from online big code resources. It can be further enriched by an organization using it. The goal is to obtain a program P_{out} that best matches the combination of content C_{in} and style S. P_{out} is retrieved from D or either generated. The essential requirement for both generation and retrieval is to obtain two independent feature embeddings that represent content and style, respectively. DUETCS first constructs a content embedding from a syntax tree representation [16]. Then it discovers and removes style information through a meta-learning strategy based on Siamese networks. Meanwhile, DUETCS constructs the approximate target style embedding by averaging the style embedding of each code example P_i^S from target style S. The style embedding is constructed by focusing on both text and structure style with a novel attention-based style generation network (Section IV).

After generating content and style features for the desired transferred code, DUETCS combines them for both modes. *In retrieval mode*, DUETCS directly retrieves the best fitting candidate piece of code in the target style from the big code dataset (Section V). *In generation mode*, DUETCS generates a piece of code by encoding both feature types into the generation model, which is based on LSTM (Section VI). Both modes work independently. DUETCS obtains the results from both modes and calculates the content similarity of each with the original input code and style similarity of each result with the average target style. An overall score is calculated by averaging these two similarities for each mode. The result with the highest score will be checked for compilability and returned. The compilability check can also be ignored because our goal is to provide as much assistance as possible to user. If there are some minor grammar mistakes, user can always manually revise the code. And DUETCS can also return more pieces of candidate code to help user to find the best-effort style transfers.

IV. EMBEDDING STYLE AND CONTENT

To make transferring a given piece of code to the desired style possible, the method proposed should be able to identify and encode the target style features but also ensure that the content of the source code is retained. In this section, we discuss how to capture and separate style and content features.

A. Style Feature Embedding

Existing work restricts itself to one standard code style [8], [26], [23], and does not capture the diverse project-and programmer-dependent code styles. Approaches that can handle multiple styles require manual generation of style rules [28]. In our approach, the user provides a set of example programs that defines the desired target style. Thus, DUETCS has to learn an abstract style embedding that can be extracted from arbitrary pieces of code. To create such an embedding, we formulate a problem where code examples are classified into latent styles based on their style features. To train this classification model, we need a dataset where code pieces are assigned to latent style specifiers as their labels. A proxy for the style label can be the contributor to a specific source code.

1) Style aspects: DUETCS expands existing work by capturing more style aspects concerning text and structure beyond formatting.

Text style includes **formatting** and **naming**. **Formatting** changes the appearance of the code. For example, indentation makes code blocks distinguishable, spaces separate tokens, and new lines break down long lines and separate concepts. **Naming** style captures everything related to variable, method, or function naming, all of which are typically person and/or task specific. For example, some programmers prefer to use short names for convenience and some prefer long names for readability. Other differences are to use underlines, camel case, or synonymous descriptions.

Structure style includes **ordering** of code blocks and **control structures**. Different programmers might place the fields and methods of the same class or program in different order. For example, all the fields of a class can be put either at the beginning or the end of the class definition. Similarly, different programmers might implement different logic and control structures, e.g., `for-` or a `while-`loop, for the same purpose.

Our notion of *latent code style* refers to a closed subset of rules that captures the aforementioned style aspects and can be associated and labeled with a contributing entity, i.e., developer, guide, organization.

2) Encoding style-specific features.: The reason that previous work mainly focus on formatting is that it can be detected locally and explicitly, for example, by just checking the tokens. However, identifying a holistic latent style cannot be done with focus on local parts of the code but requires an abstraction of latent features. Our work aims to capture more global features.

One could resort to a neural network model, which automatically generates an applicable embedding. The network will learn an abstract feature embedding from the input code token sequence. However, this approach relies on large training data without leveraging any knowledge of actual style-specific features, such as the code structure or style-related tokens. With such knowledge, the model could pay more attention to the relevant parts of the input code. With the enhanced features, the model also needs less training data and could generalize better for unseen styles. Thus, we integrate an initialization step to encode the style-specific knowledge on text and structure into the embedding.

Text style embedding. To encode the text style features, DUETCS first converts the code to a token stream that retains all formatting elements [23]: whitespace, tabulation, newline, whitespace indentation increase, whitespace indentation decrease, tabulation indentation increase, tabulation indentation decrease, single quote, double quote, empty gaps between non-label nodes. To simplify this process, all tokens are one-hot encoded. When encoding variable names, we consider that names are created by individuals. Names might be heavily

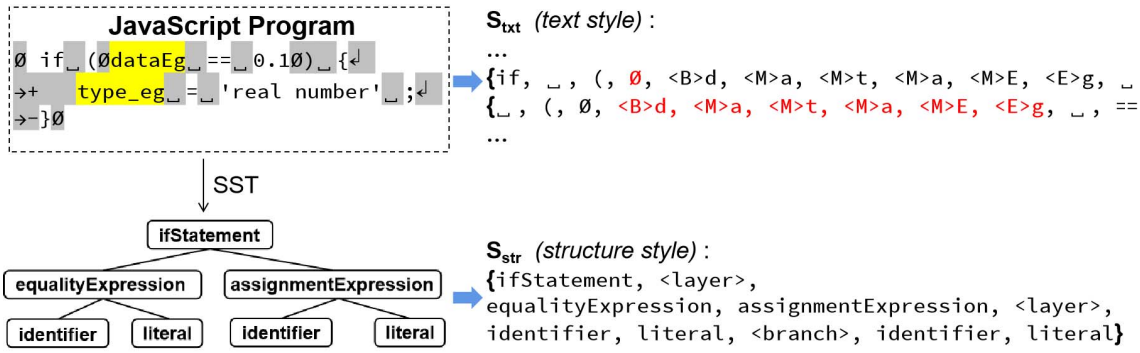

Fig. 2: Vectorization of text and structure style features

heterogeneous and new names are continuously created. It is difficult to learn and predict new or rare names with a static vocabulary that only includes seen and frequently-used tokens. To address the out-of-vocabulary problem, we adopt a character-level embedding for the names, where we also retain information of character positions. We label the character at the beginning of the word with <B>, at the end with <E> and all characters in the middle with <M>. For example, Tmp is encoded as <B>T,<M>m,<E>p.

To put formatting and naming features together, we also encode the context where they occur in the code snippet. For this purpose, we consider for each token a fixed-size window of tokens with an equal number of tokens before and after this token. For each token, we encode its whole sequence window. Consider the JS program in Figure 2. At the top right is the encoded token sequences of size 7 for the second empty formatting token and the name token dateEg. The variable name dateEg is encoded at character level, and the reserved words such as if are encoded at word level. We encode all the tokens from the code the same way and create a set of token windows S_{txt}.

Structure style embedding. Existing structure representations, such as the syntax tree or control flow graph [12], [13], [17], contain all structure elements. They represent the overall code structure rather than the style structure. To limit the encoding only to style structure, we use the simplified syntax tree (SST) [16]. The structure of SST is shown at the bottom left of Figure 2. SST shows the complete syntax structure of the code where nodes represent the construct and edges represent the relation between each construct. But it eliminates repeating and intermediate nodes that contain redundant information, which enables it to extract simpler features. We use SST because it conveys the style features of the control structures and the ordering aspect as it creates branches following the writing order of the program. To traverse the tree and encode it into the model, we convert it to a sequence. Specifically, we traverse the tree with breadth-first search, i.e., from the top to the bottom, layer by layer, and read the nodes from left to right. We use the name of the visited node as the token in the sequence. To make the encoded structure indicate the orders, we also insert special

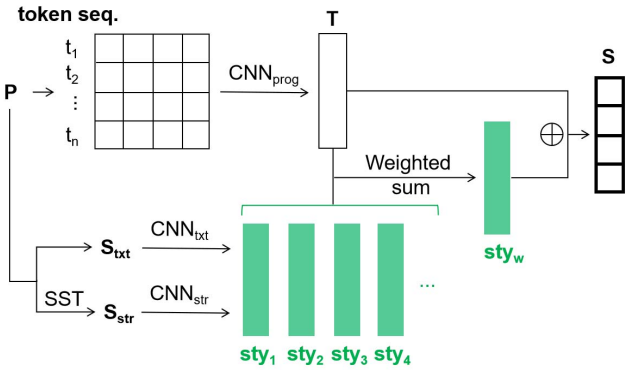

Fig. 3: SGN for constructing initial style embedding

tokens: <layer> indicates the start of a new layer, and <branch> indicates the start of a new branch. The bottom right of Figure 2 shows the generated sequence that represents the SST. This sequence of tokens S_{str} is then encoded in DUETCS as structure style features.

Style generation network. So far, we have obtained a set of text and structure style features. When generating the style embedding, we can use these features as prior knowledge to guide the model where and how much to focus attention on [30]. The encoding process is conducted by a style generation network (SGN) shown in Figure 3. We first use a CNN to encode the entire token sequence $\{t_1, t_2, ..., t_n\}$ of the program P into a vector T. Then we use two CNNs (CNN_{txt} and CNN_{str}) to separately encode text style features S_{txt} and structure style features S_{str} that we introduced before. Each component i of S_{txt} and S_{str} will be converted to a vector sty_i, which has the same dimensionality as T:

$$\mathbf{T} = \text{CNN}_{prog}(\{\mathbf{t_1}, \mathbf{t_2}, ..., \mathbf{t_n}\}), \quad (1)$$

$$\mathbf{sty_i} = \text{CNN}_{\text{X}}(S_{\text{X}}), X \in \{\text{str}, \text{txt}\} \quad (2)$$

To guide the embedding process with the extracted style features, we compute the impact contributed by each component of the style feature vector S_{txt} and S_{str} on the style of the whole program. We achieve this by taking the inner product between the code vector T and each style vector sty_i and feeding it to a softmax function. The result is used to

calculate the weighted sum of all the style feature components and produce a vector sty_w:

$$p_i = softmax(\mathbf{T}^T \mathbf{sty_i}), \tag{3}$$

$$\mathbf{sty_w} = \sum_i p_i \mathbf{sty_i} \tag{4}$$

This weighted vector and the input token vector T will be aggregated by a dense layer to generate the style embedding, this fully-connected layer connects all the preceding inputs and produces a vector in the desired dimension:

$$\mathbf{S} = dense(\mathbf{sty_w} + \mathbf{T}) \tag{5}$$

3) Training the Classification Model: To build an encoder that can automatically generate the style embedding of arbitrary code, we train the style embedding as a style classification problem. Then we can take the encoder of the classification model as our style embedding encoder. In a big code repository, typically there are many different styles (contributors), e.g., ˜85 million in GitHub, but relatively few code pieces per style. The ratio of training instances to the number of different labels is very low and unsuitable for deep learning [35]. Thus, we transform the original classification task to one with fewer categories and more data per category. We leverage Siamese networks, which is a metric-spaced meta-learning approach [14]. Instead of predicting the style label of a program, this approach transforms the original task into the task of predicting whether two programs are same style. This way, we solve a binary classification problem with large numbers of labeled training data as we can randomly pick any code pair from the big code as one training instance. Although in practice, it is possible that the coding style of one contributor is not consistent. But under the condition of lacking supervised training data, the approach can help us to the greatest extent possible to build a relatively effective style encoder due to the large scale of big code.

The structure of the Siamese network is shown in Figure 4. The green color indicates the steps of style feature generation. There are two identical neural networks with sharing weights. For each input code pair, the neural network generates a feature embedding for each code piece individually. Then their similarity is calculated for prediction. The input of the network is a pair of programs. We first use the SGN to construct the initial style embeddings S_1 and S_2 for each input program, respectively. Positive input samples with label 1 include two programs from the same style. Negative samples with label 0 include two programs with different styles. The model calculates the distance D between the style embeddings S_1, S_2 to measure their similarity. Then a dense layer transfers D to a scalar to calculate the similarity score with a sigmoid function. We use cross-entropy as the loss function. We can use the trained SGN to generate an approximate style embedding for any program.

4) Model fine tuning: Since Big Code contains large numbers of latent styles but relatively small numbers of samples for each style, even the binary classification model might still highly overfit. We further fine-tune the model to obtain more

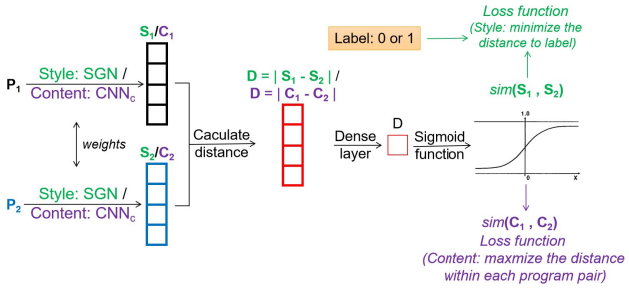

Fig. 4: Siamese feature network

general and accurate style embeddings. To do this, We employ the encoder from the classification task on a new task. We generate a support set by randomly drawing several new style categories that are not in the training set. Then we randomly pick one program from the support set. The model needs to figure out which style category it belongs to by comparing its style embedding to other code samples. This way, we can enhance and evaluate the model's ability of learning to learn by improving its performance on more data.

Specifically, we use the pre-trained model to generate the style embedding for the samples in each category in support set. We calculate a mean embedding of all samples as the style feature μ for each category. We also generate the style feature for the query (q). Assuming there are n categories in the support set, to judge which category the query belongs to, we use the softmax function:

$$\mathbf{p} = softmax(\mathbf{Wq} + \mathbf{b}) \tag{6}$$

where W and b are vectors with m dimensions. Each component of vector p represents the probability that the query belongs to a specific category. To initialize the training, we set the initial value of b to 0, and the initial value of W to the cosine similarity of the style feature between query and each category. Therefore, the probability vector p becomes:

$$\mathbf{p} = softmax\left(\begin{bmatrix} sim_{cos}(\mathbf{w_1}, \mathbf{q}) + b_1 \\ sim_{cos}(\mathbf{w_2}, \mathbf{q}) + b_2 \\ \dots \\ sim_{cos}(\mathbf{w_m}, \mathbf{q}) + b_m \end{bmatrix}\right) \tag{7}$$

We use w_i to denote each component of W. The initial value of w_i is the style feature of each category - μ_i. We use cross-entropy as loss to train the model on the support set to fine-tune the parameters in CNN and the value of W and b. This way, we obtain an improved encoder to create style embeddings. Because the support set is usually small, we also add entropy regularization H to the loss to avoid overfitting. Let (x_j, y_j) be the labeled sample in the support set, the final loss function L is:

$$\mathbf{L} = \sum_j CrossEntropy(\mathbf{y_j}, \mathbf{p_j}) + \overline{\mathbf{H(p)}} \tag{8}$$

$$\mathbf{H(p)} = -\sum_i p_i \log p_i \tag{9}$$

B. Content Feature Embedding

Similar to the style embedding, we also explored how to construct content embedding that excludes style information.

To accomplish this, we leverage the program representation used in recent work of cross-language retrieval [16], [15]. This representation is used to retrieve code with similar functionality in a different language. In the same spirit, it could be used to distinguish code with similar functionality from the same language with different styles. It is a feature vector including structural and textual features. Each component in the vector is either a structural or textual feature element. However, this representation still includes some style aspects, such as naming elements and control structures hidden in the syntax tree paths. We aim to remove these style information to avoid interference of content and style. To adapt the code representation used in cross-language retrieval to the content embedding in our work, we use convolutional neural networks (CNN_c). Because the initial feature representation is a large two-dimensional vector, CNN can reduce the dimensionality while keeping the significant information. The CNN used in DUETCS comprises a convolutional layer and a max-pooling layer. The input is the initial representation, the output is a one-dimensional content embedding C.

To train the CNN_c and make C contain as little style information as possible, we again use the Siamese network shown in Figure 4. In the figure, the purple color represents the pipeline of content feature construction. The main idea is: for different programs written in the same style, i.e. by the same contributor, the similarity of their content embeddings should be as low as possible. If the embeddings have more similarities, it means there is more style information hidden in them. To achieve this, we first use CNN_c to generate the content embedding. Every input program is processed by the same CNN and shares the same weights. Then using the same method in Section IV-A3, we calculate the similarity between each pair of content embeddings from the same contributor. The training target is to maximize the similarity between each content embedding. CNN_c and the dense layer in Siamese network can be updated by training this whole network on Big Code. We can use the trained CNN_c to generate an approximate content embedding for any new program.

V. RETRIEVAL MODE

Our approach for style transfer is to leverage the huge amounts of well-maintained but redundant open-source programs online. In case the desired piece of code or similar code already exists in the database, we want to find it. In the retrieval mode, DUETCS uses the features to directly retrieve the piece of code with similar style as the desired style from the code database. Existing code retrieval tools [21], [15] cannot be used to retrieve code with different target style.

Database Indexing. To apply code retrieval, we first need to prepare and index the code database. For each program in the database, we construct the style feature embedding S and content feature embedding C using the feature extraction methods introduced in Section IV. For each style - in reality one can use the project or contributor to group code with the same style-, we calculate and store the mean style embedding as the style feature. This allows us to efficiently identify groups of code that are relevant to a user-specified target style. For content features, we use the index structure PBI from previous work, which maps abstract paths from the syntax tree to enable fast code retrieval [16], [15].

Retrieval Phase. The retrieval process consists of two steps. Our search strategy first filters the database for the fitting style and then for the content. In theory, we could also start with the content features and then filter based on style. However, if we first look for similar content, we might obtain too much noise, i.e., programs with similar functionality in the retrieved list, upon which checking the style will not serve as an effective filter. Given the pieces of example code in the target style, DUETCS generates the style embedding for each example and calculates the average embedding S_t. Using S_t, DUETCS identifies the codebase with the mean style embedding that is most similar to S_t. It is possible to also consider top-k similar codebases. Or the user might verify whether the retrieved codebase has the appropriate style.

In the second step, DUETCS generates the content embedding for the input program, which serves as the query to retrieve the code piece with most similar content from the codebases retrieved in step one. A ranked list of the most similar code pieces will be the result of this step.

The retrieval mode and the generation mode produce results independently and simultaneously. The result from which mode will be chosen as the final output is based on the aggregated content and style similarity with input and target, respectively.

VI. GENERATION MODE

As there is no guarantee that the desired piece of code exists in the database, we also need a generation mode to automatically generate the code that approximates desired style based on the given source code. Similar to the retrieval mode, DUETCS first constructs the style embedding S_t for the target style and content embedding C_{in} for the source code. Then it uses a generation model to output a piece of code based on C_{in} and S_t. As a program can be sequenced in tokens, we leverage long short-term memory (LSTM) to build this model [19]. Normally, LSTM is an end-to-end model that generates the output sequence with the maximum probability on the condition of the input sequence with the same type. The input is the original code, and the output is the transferred code. However, there are several problems in using this LSTM architecture. First, when using Big Code, we cannot assume to have a perfect mapping between different pieces of code in different styles, which leads to a lack of training data. Second, we need to train a different LSTM for each input-output style pair, which brings computation overhead. Thus, we propose a new LSTM architecture by changing its input. Because LSTM encodes any length input into a fixed dimensional vector that is then decoded to the output, its input does not have to be the raw code. We use the combination of source content feature and target style feature as input. We only need to encode them into the same dimensional vector.

This way, we transform the code-to-code generation to generating code based on its style and content features. The biggest advantage is that we do not need labeled training data in form of code-to-code mappings. For each code piece in the database, we construct its style and content embedding, and encode them as the input of the generation model, then use the code itself as the output (label) to train the model. During the style transfer step, DUETCS constructs the target style embedding S_t and source content embedding C_{in}. Then it uses the trained model to generate the corresponding transferred code. This time the code will be in a different style as given by the input examples.

Specifically, the training goal is to maximize the probability of the program y given its content feature C_{in} and style feature S_t. y represents the token sequence of the program. The joint probability can be described as

$$\log p(\mathbf{y}|\mathbf{S_t}, \mathbf{C_{in}}) = \sum_{i=0}^{n} \log p(\mathbf{y_i}|\mathbf{S_t}, \mathbf{C_{in}}, \mathbf{y_0}, ..., \mathbf{y_{i-1}}) \quad (10)$$

where n is the length of the token sequence, y_i is the one-hot encoding as described in section IV-A2. To improve the compilability, the formatting tokens are also included. Each LSTM cell takes a fixed length hidden state (memory) h_i, which will be updated in the next cell. In each step, the LSTM cell takes the current token and memory as input, then outputs the next token and updates the memory. The output then becomes the input of the next cell. The update phase is carried out as follows:

$$\mathbf{h_{i+1}} = \text{LSTM}(\mathbf{h_i}, \mathbf{x_i}), \qquad \mathbf{x_i} = softmax(V\mathbf{h_i}) \quad (11)$$

where x_i is the embedding of each token y_i, V is the weight that will be updated during training. The initial input of the generation model is the concatenated embedding of them:

$$\mathbf{x_{-1}} = emb_{in}(W_s\mathbf{S_t} + W_c\mathbf{C_{in}}) \quad (12)$$

where W_s and W_c are the weights for style embedding and content embedding respectively. For the training process, we use stochastic gradient descent to optimize the sum of the log probabilities in equation 10. During the prediction phase, the generation model outputs a complete program that captures the most target style feature.

VII. EXPERIMENTS

We conducted extensive experiments to show the potential of DUETCS and its variations for different languages (C++, JavaScript, Java), different style-related tasks (code style transfer, formatting, and method name prediction), and different style aspects (formatting, naming, and structure style). As there is no existing code style transfer system, we compare DUETCS with existing work that is specialized for individual style aspects, i.e., formatting style learning and code representation models. Finally, we conduct a micro-benchmark to gain an in-depth understanding of the system's performance. We published our code and model on our repository[1].

Datasets. We have two different types of datasets. One is for training all the models and serving as the database for retrieval, another is testing datasets that are on small scale and with testing labels. For the first dataset type, we use the **Public Git Archive** (PGA) database, which covers more than 260,000 bookmarked GitHub repositories [22], [1]. We fetch all projects written in three popular languages (JavaScript, Java, C++) and clean them by removing duplicate and unparsable files. We split all files into functions as training samples and remove duplicates at this stage. We obtain a 239GB dataset with 13,652 projects and around 2,490 functions for each project. The average number of lines of code is 37. For each language, we train the siamese feature network together with SGN/CNN_c end-to-end and train the generation model independently. The following datasets are for testing. Since there is no existing labeled dataset for code style transfer, we create our own labeled dataset **Codeforces** based on codeforces [2], a website that provides programming problems with solutions submitted by different members. An arbitrary pair of code samples solving the same problem is a pair of labeled data.We only collect code from the top 20 rated members for quality. We obtained 20,721 C++ code samples in total. **Jsformat** is a JavaScript dataset used by a formatting baseline [23], which includes 19 top-starred JS repositories on GitHub. **Java-small** is a Java dataset used in baselines [10], [12], [13], [11], [36]. It encompasses 11 Java projects with about 700K code samples. Since most of our testing data are also from GitHub, which is the same resource for the training dataset/retrieval database, we removed all test data from the PGA database to avoid data snooping. As it is difficult and requires large human resources to label the style of all datasets, we assume each project represents one unique formatting style. Although it is a rough labeling strategy and will introduce noise, it is the best effort solution to evaluate our system considering both effectiveness and the available resources. In real-world cases, users can freely choose their own labeling strategy. For example, we also suggest contributor and repository as possible alternative labels.

A. Performance on code style transfer (C++)

We use the **Codeforces** dataset to evaluate DUETCS on transferring code style. For each pair of styles, we randomly pick 10 matched code sample pairs. From each pair, we pick one member as input, another one as the code for the target style. DUETCS constructs the style features based on all the code samples in target style. Then we switch the roles in this pair and repeat. This way, for each pair of styles, we generate 20 tasks.

Metrics. We explore the potential of DUETCS on assisting style transfer with regard to four aspects:

- **Reference accuracy** (A_R): the percentage of results that match the ground truth. As semantically equivalent programs might have different text/structure, a low A_R system does not necessarily indicate poor performance. It is possible to use the code fragments more accurately in a contextualized setting in live coding even though the results of the approach slightly diverge from the ground truth. To measure this potential, we analyze the following metrics.

[1]https://github.com/LUH-DBS/Binger/tree/main/DuetCS

TABLE I: Performance of variations on DUETCS

Method			A_T	A_S	A_C	A_R	Data needed	Data tested	Avg. LOC
DUETCS (G)	Style features	S+A	59.6	60.4	21.3	2.3	524	3800	47
		T+A	70.8	52.7	21.7	2.9			
		T+S	71.8	61.2	27.1	5.2			
	Content features	C2S	38.6	34.4	3.3	0.3			
		CTrans	31.6	37.8	2.4	0.0			
		RPT	68.9	62.8	10.1	1.7			
	Default settings		72.9	66.9	31.6	7.0			
DUETCS (R)			67.7	57.6	48.1	10.2			
DUETCS			71.8	64.8	55.8	13.1			
DUETCS (Ideal)			**77.6**	**71.3**	**64.2**	**15.5**			

TABLE II: Value of results with regard to compilabilty

	Compilation checking			
	DUETCS		DUETCS (Ideal)	
	ON(default)	OFF	ON(default)	OFF
Text accuracy	71.8	65.6	77.6	67.3
Structure accuracy	64.8	59.8	71.3	67.7

- **Computational accuracy** (A_C) [29]: the percentage of programs that can be compiled and produce the same output as the ground truth reference when given the same input. We can directly test A_C on codeforces website.
- **Text accuracy** (A_T): we use BLEU [25] to measure the text-level similarity with ground truth, and take the average similarity of all test cases as the text accuracy.
- **Structure accuracy** (A_S) [27]: we measure the similarity of the ASTs between the prediction and the reference by calculating their largest common tree prefix (LCP). Then we take the F1 score as the accuracy on the structure level.

Baselines. In absence of direct competitors, we evaluate **variations** of DUETCS: only generation mode - **DUETCS (G)**, only retrieval mode - **DUETCS (R)**, and the default **DUETCS** containing both. For **DUETCS (G)**, we test different feature representations. For style features, the default **DUETCS** uses both text style (T) and structure style (S) as prior-knowledge to guide the generation of the attention-based embedding in SGN (A). We further test variations by removing one component at a time: only encode S in the SGN (**S+A**), only encode T in the SGN (**T+A**), and simply encode T and S without SGN (**T+S**). We also compare our content representation to three code representations: AST-based model **C2S** [11], Transformer-based model **CTran** [36], and the representation for cross-language retrieval **RPT** [16]. We also report **DUETCS (Ideal)**, which always correctly chooses between the result of the generation and the retrieval mode. To simulate this ideal scenario, DUETCS directly compares the results returned by both modes with the ground truth and chooses the one that has the higher similarity with the ground truth as the final output.

Runtime. We ran our experiments on a PC with an Intel Xeon E5-2650 v2 2.60GHz CPU and an NVIDIA Tesla K40m GPU. The total offline training time is around 3 days. During the online phase, there will be an overhead in the order of one minute to learn the target style from the examples. However, this process happens only once per use case. The learned style features are saved and can be reused in future style transfer tasks. The final prediction step for a piece of code happens within a second. Therefore, DUETCS is practical for real-world use.

Results. The dataset statistics and the results are shown in Table I. We inspect the text accuracy and structure accuracy of all the outputs and found that DUETCS can produce results that are around 70% similar to the ground truth at both text and structure level. It shows that DUETCS can provide significant assistance to user on the style transfer task. After inspecting the compilation checking module, we found that DUETCS achieves 55.8% computational accuracy, which is a practical metric for a code generation system [29]. This result shows that more than half of the output code are compilable and implement the same function as the input code. The user can use this check as an optional layer of the pipeline to guarantee grammar correctness. We also inspect the A_T and A_S of the outputs that are not compilable in Table II. We found that even the non-compilable outputs display around 60% similarity to the ground truth, which means even if DUETCS cannot always produce grammar-correct code, it can still provide valuable information to help user to transfer code style. We also further checked if the output can be exactly the same as the ground truth. Notice, that generally the task of generating the exact same code as ground truth is very hard, especially when the code length is rather long (˜47 lines). Reference accuracy is not a solid indicator of performance. The default system outperforms all other variations and outputs 13.1% results that exactly match the ground truth on 3,800 samples. To put this number in context, consider the performance of the code generation system TransCoder, which achieves less than 10% A_R on average for program translation [29]. In an ideal scenario, DUETCS can achieve 2.4% higher A_R and 8.4% higher A_C. All the aforementioned performance is achieved fully unsupervised. For each prediction, we only need to take all the raw code samples in the target style to construct the style representation. There are no additional labels to these code samples. Moreover, DUETCS only uses 524 samples for each prediction on average, which is a small amount of data compared to data needed for general supervised machine learning.

The A_R and A_C of the retrieval mode are higher than the generation mode, mainly because machine-generated code cannot always conform to rigorous programming language grammar. This is on par with results obtained for translation discovery [16]. On the other hand, generation mode leads to higher textual and structural accuracy, A_T and A_S, suggesting that it can produce better approximations of the desired code than retrieval mode. While the retrieval mode might find a more accurate result from the database when the generation mode failed to generate a proper transferred code, generation mode can generate better approximations when the target code does not exist in the database.

Next we analyze the influence of different feature rep-

resentations on DUETCS's performance. If we exclude text style (S+A) or structure style (T+A), the performance is generally worse than when excluding SGN (T+S). This shows that it is reasonable to focus on the text style and structure style as they are more significant. Furthermore, features without structure styles (T+A) perform better than without text styles (S+A), which shows text style plays a more significant role in identifying coding style. Comparing the results between style features and content features, we can also infer that content features are more important for code generation. Another reason could be that the variations of content features are dedicated to including all the content features while the variations of style features only include part of the style features. RPT, which we used as the basis of our default content features, achieves the best results because it represents mostly the content information to perform cross-language search. CTrans performs the worst maybe because it uses a pointer network that encodes the original naming style information in the features.

B. Performance on JSformat (JavaScript)

We further evaluate DUETCS on the JSformat dataset. Because we lack the ground truth for this dataset, we alter the style transfer task to style recovery. Given an input code with some inconsistent style features compared to other code samples in this style, DUETCS reconstructs the input code by fixing the style. We evaluate DUETCS's performance with regard to three style aspects: formatting, naming, and structure. For formatting style, we perform the same task from STYLE-ANALYZER [23], which is to predict all the formatting tokens in a piece of code. For naming style, we mask the method name as input and check whether the reconstructed code can recover this name correctly. For structure style, we randomly switch the order of two code blocks and check whether DUETCS can reconstruct the original order.

Metrics. In this experiment, we are comparing to an approach that is only able to predict formatting tokens. Thus, the metrics we used in the last experiment, which measure the similarity between the whole code snippets, cannot apply to this task. We use two other metrics to measure the performance:

- **Precision (Prec)** is the percentage of correct predictions of all the predictions, which indicates the performance on inconsistent style correction.
- **Prediction rate (PredR)** [23] is the percentage of actual predicted tokens of all formatting tokens that need to be predicted. It indicates how often the system makes predictions, i.e. the performance on inconsistent style detection.

Baseline. We compare DUETCS to the only unsupervised formatting system STYLE-ANALYZER [23]. As proposed in their paper, we split each repository's files into two groups that contain 80% and 20% of all files. The first group is used for learning the style, the second group serves as test data.

Results. The upper part of Table III shows the results. In terms of functionality, STYLE-ANALYZER can only learn formatting style and is not comparable with DUETCS on naming and structure style. Besides, it cannot generate the whole code snippet. It takes the token sequence that centers on each formatting token as input to predict this formatting token. Thus, in the table, the amount of data used/trained and the amount of data tested in STYLE-ANALYZER is at the granularity of token sequences and tokens while in DUETCS is at the granularity of code samples. For STYLE-ANALYZER, the average line of code (LOC) in each case is less than one line and the location of the tokens to be predicted is already known. On the contrary, DUETCS can directly work at the whole code level and address multiple styles in fully unsupervised way. From these aspects, DUETCS is more functional.

TABLE III: Performance on JSformat&Java-small dataset

	Method	Formatting		Naming		Structure		(train/ test)
		Prec	PredR	Prec	PredR	Prec	PredR	
JSformat	**DUETCS**	89.2	71.9	48.7	100	91.9	87.3	1,269/ 6028
	DUETCS (G)	91.2	58.7	33.5		**96.6**	76.5	
	DUETCS (R)	**97.2**	43.6	21.1		90.6	63.2	
	DUETCS (Ideal)	93.1	75.4	**54.6**		94.7	**91.2**	
	STYLE-ANALYZER	92.5	**91.6**	-		-		139,615/ 663,171 token seq.
Java-small	**DUETCS**	93.1	62.0	53.9	100	81.2	93.7	66,512/ 721,280
	DUETCS (G)	92.1	53.6	39.4		80.9	78.8	
	DUETCS (R)	89.1	31.5	20.3		**97.8**	45.7	
	DUETCS (Ideal)	**94.5**	**68.3**	**59.7**		86.3	**94.2**	
	ConVAttention	-		50.3	100	-		665,115/ 56,165
	Path+CRFs			8.4				
	C2V			18.5				
	C2S			50.7				
	CTrans			54.9				

In terms of performance, for both formatting and structure style, DUETCS can recover around 90% of the original style. Both generation and retrieval modes can achieve good precision, while their ability to detect style inconsistency is worse than the full system, which shows combining two modes is a proper strategy. In the ideal scenario, when DUETCS can correctly recommend the best result, its precision on formatting style is slightly better than baseline (0.6%). Note that, the ideal mode in some cases, such as the precision of formatting, performs worse than other modes. The reason is that ideal mode chooses the globally best solution with regard to ground truth. Thus, its result might miss some aspects of style not optimally and perform worse in that regard. On the other hand, the PredR of DUETCS is about 20% lower than STYLE-ANALYZER because the prediction locations are unknown to DUETCS but known to STYLE-ANALYZER. Yet, DUETCS can well generate the whole code snippet only based on its features in 70% of the cases. For structure style, as only the order is changed, DUETCS does not need to learn other features except the order of code blocks, it can achieve around 90% PredR. For naming style, because the location of method name is fixed, the PredR is always 100%. The Prec for recovering this style inconsistency is much lower than other two styles as method names are more complicated than formatting and code blocks order. Considering the code length is relatively long (>86 lines), it is more difficult to generate the

exact same code. However, in an ideal scenario of DUETCS, it can still exactly recover more than half of the method names.

C. Performance on Java-small dataset (Java)

We further evaluate DUETCS on Java-small dataset. Same to JSformat dataset, we perform original style recovery regarding three aspects: formatting, naming, and structure.

Metrics. As the baselines in this experiment can only address one style aspect, we use the same metrics as in Section VII-B.

Baselines. We compare DUETCS to existing code representations to evaluate the effectiveness of our feature representation. The Java-small dataset is used with the following feature representations for method name prediction: **ConVAttention** [10], **Path+CRFs** [12], **C2V** [13], **C2S** [11], and **CTrans** [36]. We use them as the baselines. However, they ignore the formatting tokens when constructing the features and they cannot generate the whole piece of code, so they are unable to perform formatting and structure style recovery. For name prediction, they need training samples to supervised learn the naming mechanism. The input is the method body and the method name is the corresponding label.

According to the corresponding papers, each baseline takes 10 projects as training, and 1 project as test data. For each test code sample, they predict the method name. DUETCS only needs the code in the same style as examples and does not need training data.

Results. Functionally, DUETCS is unsupervised and covers more style aspects compared to baselines. On contrary, the baselines can only address Naming style and require a large amount of training data. The experiment results are shown in the lower part of Table III. Similar to the experiments on the JSformat dataset, DUETCS performs well on formatting and structure style. For naming style, DUETCS is around 3% better than the state-of-the-art non-transformer representation **C2S**, and in ideal scenario 9% better. The possible reason is DUETCS also encodes the writing style information into the features, while other representations that are AST-based neglect the writing style as it does not affect the semantics. Considering DUETCS is more advanced than all the baselines function-wise, the performance improvement on only naming style is already significant. Compared to the transformer-based method **CTrans**, DUETCS achieves similar result, and 4.8% better ideally. We can infer that our representation can also well learn the code context. Moreover, it shows our character-level token encoding strategy can learn out-of-vocabulary well compared to pointer network used in **CTrans**, but we do not need any task-specific labeled dataset. These experiments show that DUETCS applies to multiple types of programming languages, such as static languages (C++, Java) and dynamic languages (JavaScript). Besides, the performance on Java-small dataset is slightly better than on JSformat dataset. From Table III, we can see that the possible reason is that for Java-small dataset, there are much less training data than Java-small dataset. It means that Java-small dataset includes more examples for the system to learn the specific style features.

TABLE IV: The contribution of each mode to the final result

Scenario	Proportion of recommended results	
	from Generation mode	from Retrieval mode
Actual	75%	25%
Ideal	68%	32%

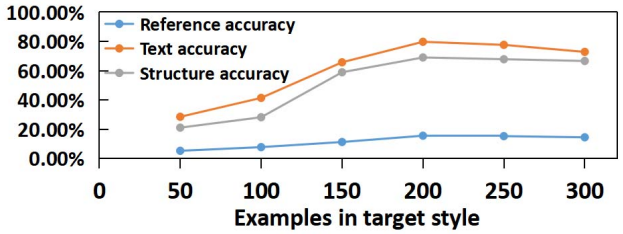

Fig. 5: Influence of the number of examples

We will further study how the number of examples can affect the results in our micro-benchmark (Section VII-D).

D. Micro-Benchmark and limitations

To further explore the practical potential of our approach on style transfer task, we analyze several components of DUETCS and investigate its limitations.

Two modes mechanism. In our experiments, DUETCS automatically chooses one result from the two modes. The returned result is chosen based on its content similarity with the input code and style similarity with the target style. We evaluate the reliability of this result recommendation by comparing the results with the ground truth. In Table IV, we show the proportion of the actual final recommendations in the experiment from Section VII-A compared to the choice of the ideal system. Note that ideal always chooses the result that is closest to the ground truth solution, while our default approach chooses the solution that is similar in terms of content to the input and similar in terms of style to the average target style. In both scenarios, the generation mode produces up to three times more suitable results than the retrieval mode. The generation mode naturally produces code that is similar to the source code, while the retrieval mode depends on the existing code pieces in the database. 18% of the time, the choices differ and as seen in Section VII-A the default system can fall behind. Thus there is still room in improving the combined effort of both modes.

Influence of the examples. DUETCS requires example code in the target style to construct the style features. In Figure 5, we show how the accuracy changes for different numbers of randomly picked examples. First the accuracy significantly increases with the number of examples. However, after a certain number of examples (around 200 in this experiment), the accuracy starts to gradually drop. A possible reason is that more examples increase the chance of adding style inconsistencies. Thus there is room for improving the performance by ensuring consistency among the given examples.

Case study and limitations. To better understand the limitations of self-learning and show how DUETCS can help in

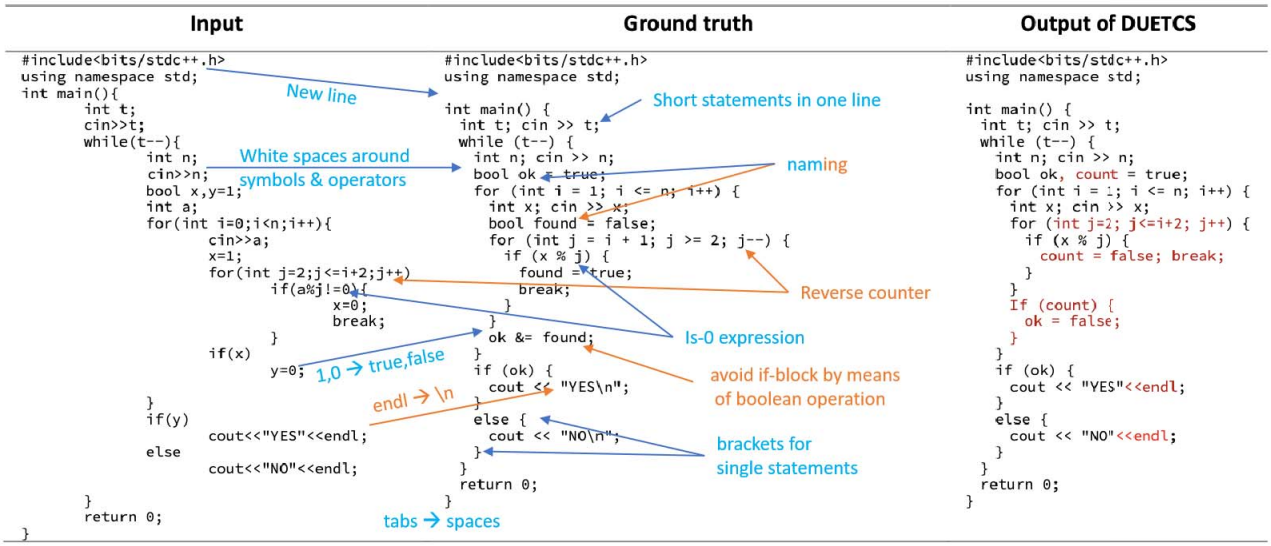

Fig. 6: Example of code style transfer and failed cases (arrows show the style differences between source and target, blue color shows that output captured the desired style and orange shows what was missed by the output)

real-world cases, we consider the example in Figure 6 where system failed to cover all style elements. The first column is the input code in source style, the task is to transfer it to the target style as shown in the second column. Comparing their style, we annotate the differences in the target style in the figure. The third column in Figure 6 shows the output of DUETCS. We mark the mistakes in red color. We studied further aligned examples from the two styles depicted in Figure 6 and identified the following common cases of failure.

- Wrong variable names (`count` in Figure 6): The term `found` is a rare name, which makes it hard to predict.
- Not counting backwards in loops: As this style has algorithmic features, it is not represented in our style feature.
- Does not use bool value operations: Also this type of style is highly convoluted with algorithmic steps and is not captured.
- Not changing `<<endl` to `\n`: The target style is not consistent. We found other examples in the target style that use `<<endl`.

In summary, self-learning captures style information as long as there is no strong convolution with algorithmic aspects and the style aspects are not singletons among the given style examples. Further, it is crucial that the provided style examples are consistent with the desired target style. However, unlike rule-based tools that require fixed pre-defined style rules, novel representations learn arbitrary target style fully automatically only from the code examples showing the significant potential of learning-based tools for stylization assistance.

VIII. Conclusion

In this work, we explored the potential of learned representations to stylize code and proposed a novel approach that leverages code generation and retrieval, and explored its potential on assisting code style transfer. For both modes, it constructs two independent code representations - style and content features. This way, DUETCS retains the content features of the input, learns the target style features, and combines them to produce the output. To the best of our knowledge, this is the first attempt of implementing an unsupervised learning approach for a holistic style transfer. DUETCS captures more relevant style aspects compared to rule-based techniques. Self-learning is applicable to multiple programming languages and does not need human intervention. It can provide user with more valuable information compared to existing baselines, such as formatting systems and variations with state-of-the-art code representation models. Although it cannot always produce the perfect transferred code, the results show that the output can still capture enough style information to assist the user in stylization. We believe that if DUETCS can be included within an IDE so that stylization suggestions are under the control of the programmer, the results of this type of style inference can be useful to the user.

There are several potential research directions in the future. First, for learning the style features from the code examples, it would be helpful to have a selection mechanism that automatically selects the most representative, consistent, and covering examples. Further, it would be interesting to explore a more symbiotic combination of the two modes that trades off accuracy and recall. Finally, it would be useful to explore methods that filter out non-compilable code based on grammar features to reduce the instrumentation overhead.

Acknowledgment

This work was funded by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A).

REFERENCES

[1] Public git archive. website. https://github.com/src-d/datasets/tree/master/PublicGitArchive [Online; accessed 15-May-2020].

[2] Codeforces. https://codeforces.com/, 2019. [Online; accessed 22-Oct-2019].

[3] Apple's ssl iphone vulnerability. https://www.theguardian.com/technology/2014/feb/25/apples-ssl-iphone-vulnerability-how-did-it-happen-and-what-next, 2021. [Online; accessed 01-Oct-2021].

[4] Apple's ssl/tls bug. https://www.imperialviolet.org/2014/02/22/applebug.html, 2021. [Online; accessed 01-Oct-2021].

[5] Oracle code conventions for the java. https://www.oracle.com/java/technologies/javase/codeconventions-contents.html, 2021. [Online; accessed 01-Oct-2021].

[6] Pep 20 – the zen of python. https://www.python.org/dev/peps/pep-0020/, 2021. [Online; accessed 01-Oct-2021].

[7] Ghazi Alkhatib. The maintenance problem of application software: An empirical analysis. *J. Softw. Maintenance Res. Pract.*, 4(2):83–104, 1992.

[8] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE), Hong Kong, China, November 16 - 22, 2014*, pages 281–293. ACM, 2014.

[9] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[10] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2091–2100. JMLR.org, 2016.

[11] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[12] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 404–419. ACM, 2018.

[13] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, 2019.

[14] Aurélien Bellet, Amaury Habrard, and Marc Sebban. A survey on metric learning for feature vectors and structured data. *CoRR*, abs/1306.6709, 2013.

[15] Binger Chen and Ziawasch Abedjan. Interactive cross-language code retrieval with auto-encoders. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021.

[16] Binger Chen and Ziawasch Abedjan. RPT: effective and efficient retrieval of program translations from big code. In *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021*, pages 252–253. IEEE, 2021.

[17] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net, 2018.

[18] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Reading beside the lines: Using indentation to rank revisions by complexity. *Sci. Comput. Program.*, 74(7):414–429, 2009.

[19] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.

[20] Ben Liblit, Andrew Begel, and Eve Sweetser. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2006, Brighton, UK, September 7-8, 2006*, page 11. Psychology of Programming Interest Group, 2006.

[21] Erik Linstead, Sushil Krishna Bajracharya, Trung Chi Ngo, Paul Rigor, Cristina Videira Lopes, and Pierre Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Min. Knowl. Discov.*, 18(2):300–336, 2009.

[22] Vadim Markovtsev and Waren Long. Public git archive: a big code dataset for all. In *ICMSR*, pages 34–37, 2018.

[23] Vadim Markovtsev, Waren Long, Hugo Mougard, Konstantin Slavnov, and Egor Bulychev. STYLE-ANALYZER: fixing code style inconsistencies with interpretable unsupervised algorithms. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, pages 468–478. IEEE / ACM, 2019.

[24] Robert C. Martin. *Clean Code - a Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009.

[25] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*, pages 311–318. ACL, 2002.

[26] Terence Parr and Jurgen J. Vinju. Towards a universal code formatter through machine learning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, pages 137–151. ACM, 2016.

[27] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 1139–1149. Association for Computational Linguistics, 2017.

[28] Steven P. Reiss. Automatic code stylizing. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pages 74–83. ACM, 2007.

[29] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[30] Tao Shen, Yi Mao, Pengcheng He, Guodong Long, Adam Trischler, and Weizhu Chen. Exploiting structured knowledge in text via graph-guided representation learning. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 8980–8994. Association for Computational Linguistics, 2020.

[31] Diomidis Spinellis. elyts edoc. *IEEE Softw.*, 28(2):104, 2011.

[32] Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Program. Lang.*, 4(3):143–167, 1996.

[33] Christoph Treude and Martin P. Robillard. Understanding stack overflow code fragments. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 509–513. IEEE Computer Society, 2017.

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[35] Yaqing Wang, Quanming Yao, James T. Kwok, and Lionel M. Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM Comput. Surv.*, 53(3):63:1–63:34, 2020.

[36] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.