Exploring Code Style Transfer with Neural Networks

Karl Munson*, Anish Savla*, Chih-Kai Ting*, Serenity Wade*
University of California, Santa Cruz
{kmunson, asavla, cting3, swade1}@ucsc.edu

Kiran Kate and **Kavitha Srinivas**
IBM Research

Abstract

Style is a significant component of natural language text, reflecting a change in the tone of text while keeping the underlying information the same. Even though programming languages have strict syntax rules, they also have style. Code can be written with the same functionality but using different language features. However, programming style is difficult to quantify, and thus as part of this work, we define style attributes, specifically for Python. To build a definition of style, we utilized hierarchical clustering to capture a style definition without needing to specify transformations. In addition to defining style, we explore the capability of a pre-trained code language model to capture information about code style. To do this, we fine-tuned pre-trained code-language models and evaluated their performance code style transfer tasks.

1 Introduction

Recent years have seen a sharp increase in research on code language modeling, with work ranging from pretraining large language models over code (Ahmad et al. (2021), Wang et al. (2021), Feng et al. (2020)), as well as related work examining the application of other natural language processing tasks applied to code, such as code machine translation or code generation. Szafraniec et al. (2022), Chen et al. (2021). However, accompanying this rise has been a lack of work studying how well these models understand the human notion of code style.

Code style is an integral element of code, so that many organizations regulate code style through use of code reviews, largely due to its effect on code readability. Code style is, broadly, the appearance of code to a viewer, and is thus - as much the same as writing style - something that is very individual to its writer. Due to the strict syntactic grammars governing how code is structured, many aspects of code style can be quantified automatically. Yun Peng (2021) programmatically studies the usage of Python language features across publicly available repositories.

We assert that the content of code (its functionality) and the style of code (its aesthetics) are, like the content and style of text, aspects whose latent representations can be extricated and individually manipulated through neural style transfer.

In this work, we defined an embedding space whose dimensions are extricable metrics measuring different aspects of code style. We then explored how to quantitatively analyze code style through clustering over this space, and how well code can be classified into these clusters. From there, we generated data and finetuned pretrained code language models to perform code style transfer along one aspect of code style.

1.1 Contribution

The main contributions of this work are fourfold. First, we applied clustering algorithms over code scripts embedded in a code style embedding space to find code clusters with distinct styles, as well as evaluated the quality and predictability of the discovered clusters. Second, from publicly available code repositories, we generated a suite of parallel corpora; each representing an atomic style aspect transformation, such as casing or comments. Third, using these parallel corpora, we fine-tuned pretrained code language models on sequence-to-sequence individual style transfer tasks. These models can perform precise stylistic alterations one at a time; such as modifying casing or adding comments. We also trained a joint model that can apply multiple style transfers at once. Finally, whereas the existing work around code language models focuses on the syntactic integrity of processed code, our work focuses on something novel: code style.

2 Related Work

2.1 Code Style

Due to the subjective nature of code style, the first background area of interest to address is what constitutes code style. This paper focuses primarily on style aspects of Python as a result of the large number of stylistic choices available in Python, large number of publicly available code samples, and familiarity with the language.

Many of the python language features that we focused on were selected based off of the analysis done in An Empirical Study for Common Language Features Used in Python Projects (Yun Peng, 2021). This paper tries to study the use and impact of Python language features in real-world Python projects. The authors analyze Python language features and automatically identify the use of 22 kinds of common Python language features in 6 categories in Python source code. This study is conducted over 35 popular Python projects from eight application domains, covering 4.3 million lines of code, to investigate the the usage of these language features in the project. Inheritance, decorator use, keyword arguments, for loops and nested classes are listed as the top 5 used language features.

2.2 Code Language Models

Recent years have seen the natural language processing literature dominated by the pre-train and fine-tune paradigm. In line with this, we will be primarily using two pre-trained language models for code for our experiments, PLBART and CodeT5. The use of pre-trained models will eliminate the need for extensive resources needed to train a code language model from scratch.

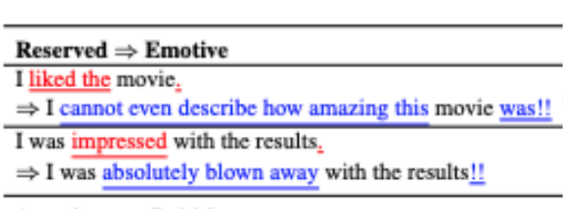
CodeT5 is implemented by an encoder-decoder model pre-trained on masked span prediction, identifier tagging, and masked identifier prediction for bimodal, programming and natural language, dual generation (Wang et al., 2021). It uses CodeSearchNet data which consist of unimodal (programming languages only) and bimodal (programming language and natural language) data on 6 programming languages. Developed by Salesforce, it is available on HuggingFace in different model sizes.

PLBART is another encoder decoder model we explored. The model is pre-trained in Java, Python, and natural language on reconstruction and denoising (Ahmad et al., 2021). Like CodeT5, it is publicly available on Huggingface in different model sizes, as well as having checkpoints finetuned on different code-related tasks.

Even though language models for code have been used for a lot of tasks, such as bug detection, code translation, code generation; to the best of our knowledge, our work is the first one to address the task of code style transfer.

2.3 Text Style Transfer

Text style transfer is the task of reformatting a natural language text from one style to a text with the same meaning in a different style, while maintaining the textual content of the source.(See figure 1) There are a number of ways to approach this task, and a significant amount of work has been done to use neural models for text style transfer (Riley et al. (2021), Tikhonov et al. (2019), Reif et al. (2021), Jin et al. (2020)). This existing work in text style transfer in natural language serves as a blueprint on how to apply the same style transfer to code.



Reserved ⇒ Emotive
I liked the movie. ⇒ I cannot even describe how amazing this movie was!!
I was impressed with the results. ⇒ I was absolutely blown away with the results!!

Figure 1: Example of text style transfer. (Riley et al., 2021)

Most approaches to this task utilize either an encoder-decoder framework or a generative adversarial network with different variations to those two basic framework, such as the one pictured below (2).

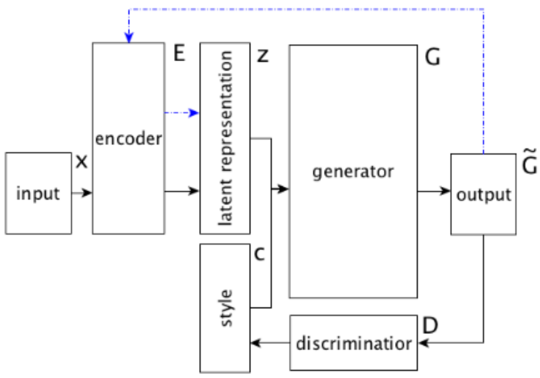


Figure 2: Example of GAN style transfer architecture. (Tikhonov et al., 2019)

3 Defining Code Styles

The core underlying topic of this research is code style, which is, most broadly, the appearance of the code. Code style is vital to the readability of code, and many organizations value code styles adhering to a centralized style guide for uniformity, or conduct code reviews to ensure that the style of written code matches a certain quality threshold. Though code style is largely subjective and very complex, we develop a framework to analyzing specific aspects of code style through quantifiable metrics, and use this framework to conduct experiments with both clustering and sequence to sequence model training. Listed below are definitions of the aspects of code style we decided to focus on:

3.1 Casing

For this metric, we consider the various kinds of casing found within code: snake case, lower and upper camel case, and lowercase. We also consider how these forms of casing appear in different kinds of identifier names, such as variable, method or class names.

3.2 Comment

Comments are vital elements of code style as they aid greatly to the readability of the code. We consider many attributes of code comments to be indicative of style; such as comment character and word length, and comment density; as well as the ratio of lines in a script that are commented, all as approximate measures of how much a script has been annotated with comments.

3.3 Docstrings

Docstrings, like comments, are written by the programmer to aid the readability of the code and are integral to code style. They serve as headers to inform any readers of the intended function, input and output arguments and types of a method, class, or module. The often contain other information as well, relating to copyright or authorship.

3.4 List Comprehensions

We additionally focus on Python language-specific features, such as comprehensions. List comprehensions are one such language feature, used to iterate and operate on the items within a data structure. A programmer's decision between using a for loop or a comprehension to do this is a Python-specific stylistic choice.

3.5 Usage of Classes

Being an object oriented language, Python allows for programmers to define their own classes. How one chooses to organize classes, the methods within them, or use of inheritance, is another hallmark of code style.

These main categories describe the main attributes of code style we chose to focus on for parallel corpora generation. We also used a wider set of attributes during clustering experiments, mostly related to the above described traits.

4 Design

4.1 Clustering

Code style is difficult to define, so we selected a host of features (See Table 1) to embed Python code into a multidimensional space, then use unsupervised clustering algorithms to identify the types of code that are most similar to each other. From these clusters, we can then define spatial buckets that represent different and distinct code styles.
In order to cluster style features we utilized the HDBSCAN(Malzer and Baum, 2020) clustering model to group code samples together based on similar counts of certain style features. HDBSCAN is a hierarchical clustering algorithm which can be approximated by a random forest to perform prediction of cluster labels on new data. The total list of features that we have are listed in the Table **??**.

4.2 Style Transfer GAN

We adopt a similar style transfer architecture to Tikhonov et al., the state of the art text style transfer on the Yelp small review sentiment transfer task. The model will be slightly different from the original paper in order to allow the pre-trained encoder decoder to run uninterrupted. The input code is concatenated with the style code and reduced using a linear layer so that it fits into the pre-trained encoder. The pre-trained encoder decoder will take this compressed input and produce a styled code sample which goes into the discriminator. The discriminator loss will be used to modify the style representation concatenated to the input.

Substitute Pre-trained Model We replace the encoder from Tikhonov et al. with the encoder of our pre-trained model CodeT5 or PLBART. We use the pre-trained encoder to generate a latent

Code Style Features for Clustering	
Feature Type	Features
Casing	Snake case variable usage ratio Snake case function name usage ratio Snake case class usage ratio Upper camel case variable usage ratio Upper camel case function name usage ratio Upper camel case class usage ratio Lower camel case variable usage ratio Lower camel case function name usage ratio Lower camel case class usage ratio
Documentation	Docstring Density Comment Density
Function/Class	Average function decorators usage count Average class decorators usage count Average class inheritance usage count
Python Features	Average list comprehension usage count Average generators usage count Average lambda function usage count

Table 1: The selected code style features which were used for clustering experiments.

representation from the input code snippet. For the model generator, we will use the decoder of CodeT5 or PLBART. The generator will output decoded code. This code is then passed to the discriminator, a 3 layer multi-layer perceptron; again following from Tikhonov et al.. The MLP will learn to represent the style of the generated code, and a style loss will be passed back to the encoder of the model.

4.3 Style Classification with Pre-Trained Models

The GAN trained model failed to generate resonable output for recognizing the style clusters, it fell back to the question that, are the pre-trained code language models even have a notion of code styles?

So we built a classification model from the pre-trained language models we tested. This was done in order to see if the pre-trained model could capture any style information from the clusters that we formed with HBDSCAN. The inputs for this model are code samples and the outputs for the model are the labels generated from the clustering experiments. We implemented 2 versions of the classifier, one for PLBART and one for CodeT5. The PLBART model is a pretrained PLBART model with a sequence classification head. The CodeT5 model uses a CodeT5 model and finetunes the generation model to produce the labels via generation and not through a classification head.

4.4 Seq2Seq Style Transfer

The style classification results showed that there is some ability for the pre-trained code language models we tested to capture style elements. As a result we constructed a sequence to sequence model to do style transformations. Instead of transferring from one style to another based on complex features, we focused on doing individual style transfer such as going from a for loop to an equivalent list comprehension. In order to do this we constructed parallel corpora for five transfer tasks, generating comments, generating docstrings, adding casing, transforming code to a class structure, and converting for loops to list comprehensions when relevant. The details for how these parallel corpora are generated is provided in the data section.

The sequence to sequence model is a pre-trained CodeT5 small instance with 60 million parameters fine-tuned on the parallel corpora. A sequence to sequence model was fine-tuned for each of the individual transformations. This was done to see if the model could be fine tuned to perform individual style transforms. This is a step towards ensuring pre-trained models can capture style transforms. In addition to this, a combined sequence to sequence model was fine-tuned on all of the parallel corpora.

This produces a multi-task model capable of doing all the transformations with only on instance of CodeT5 instead of five. To fine-tune the model with multiple tasks the dataloader was modified to randomly sample batches from each of the five tasks. The dataloader sampled proportionally to the size of the parallel corpora meaning that larger corpora were sampled more based on their size.
The combined sequence to sequence model also had a modification to its training data.

For each of the code samples a natural language prompt separated from the code by a special token was used to indicate to the model which transformation was being performed (figure:4). This allows the desired transformation to be specified during inference of the model and also allows for prompting of multiple transformations at once to be performed.

5 Data

5.1 Py150k

We used Py150k (Raychev et al., 2016), which consists of parsed ASTs parsed from Python 2.7 scripts, collected from a set of Github repositories with permissive and non-viral licenses. The set removed duplicate files and project forks, and kept only programs that parse and have at most 30,000 nodes in the AST. The set encompassed code from 5958 unique Github users, and 8422 Github repositories. The dataset is split into two parts: 100,000 files for training and 50,000 for evaluation.

5.2 BigQuery

We also used Google BigQuery for additional sources of Python code. There are up to 1.3 million python files drawn from Github with more than 2 watchers and less than 20 watchers available on BigQuery. The data is split into two parts, samples from the CuBERT(Kanade et al., 2020) dataset and new code samples which were added to BigQuery after the CuBERT paper. The files from CuBERT are already deduplicated to avoid the usage of same script again (Kanade et al., 2020). For the newer files, we did the deduplication via their paths in the intial query to remove direct copies of code samples. In order to ensure deduplication of the remaining files an MD5 checksum was calculated on the contents of each file and all duplicates were removed from the dataset. The query used was based off of the query used in the CuBERT paper and will be make available in addition to other resources used in this research.

5.3 Parallel Corpora

For the sequence to sequence transformation we generated parallel corpora for each individual style transformation for training. For all transformations excluding comments, all of the code samples had comments removed, because the AST module used for transformations strips comments; so for the transformed data to match the original, the original code also needs comments removed.

5.3.1 Casing

For casing we selected identifiers and method names using the AST module, lower-cased all of the data and stripped out underscores. This data with uncased code is the inputs and the label is the code sample with original casing. The model thus learns how to format identifier names. There are 700,000 examples in this dataset.

5.3.2 Docstring

Code samples were split using the AST module isolating methods with docstrings attached to them. The input to the sequence to sequence model has the docstrings stripped and the label is the original code with the docstring. There are 2,500,000 examples in this dataset.

5.3.3 Comments

We took code samples and parsed, then unparsed the code using the AST module. This parse strips the comments and returns the same code with only comments removed. The input is the uncommented code and the label is the original commented code. There are 700,000 examples in this dataset.

5.3.4 Classes

We took code samples with classes in them and removed the class definitions of the class and all references to self in methods. The input for this model is the code with class related syntax removed and the gold label is the original code. There are 400,000 examples in this dataset.

5.3.5 List Comprehensions

Code samples with list comprehensions in them are selected via the AST. To make the versions without list comprehensions each list comprehension is converted to an equivalent function. The input for this model is the code with list comprehensions converted to functions and the gold label is the original code. This dataset has 36,000 examples.

6 Experiments

6.1 Code Style Clustering

We collected metrics(table 1) for the scripts based off of the style features specified in the design section, and then used HDBScan (Malzer and Baum, 2020) to find and evaluate clusters of code. HBDSCAN has three parameters that we tuned to reduce the number of clusters while keeping clusters distinct. To measure cluster quality, we used Davies-Bouldin index.

In addition to measuring cluster distinctness, we also measured the predictability of the cluster results using several non-neural classifiers for testing whether the features have enough signals to build a high quality classification model since the code style could be sophisticated and difficult to characterize. We tested SVM, decision trees, random forest, naive Bayes, and logistic regression models ability to predict cluster label given the constructed style space for a script.

6.2 Style Transfer GAN

To test the viability of this model we implemented it as described in the design section with two different configurations of pre-trained model. In one case we used CodeT5 and in the other case we used PLBART as the pre-trained component. In both cases the GAN model was unable to output valid python code let alone perform the style transfer task despite tuning of parameters and troubleshooting the model. As a result of this, experiments were not continued with this architecture.

6.3 Style Cluster Classification

The failure of the Style Transfer GAN could indicate that the way we defined clusters could not be captured by a neural network. As a result of this possibility, we tested if CodeT5 or PLBART were capable of capturing code style at all. In order to test this, we tested different classifiers built on pre-trained code language models. We tested the CodeT5 small, 60 million parameters, and base, 120 million parameters. For PLBART we tested with the uclanlp/plbart-multi_task-python checkpoint on huggingface as well as "uclanlp/plbart-base". Both PLBART models have the same parameter count, but with different fine-tuning. For hyperparameters of both models, we used a learning rate of 1e-4, weight decay of 0.01, and trained for 4 epochs. These parameters were chosen as similar pre-trained classifiers used these parameters. Four epochs of training was done as validation loss was not improving much. The rest of the hyperparameters used are default huggingface parameters for each model.

6.4 Seq2Seq Style Transfer

Style cluster classification showed the ability for pre-trained models to capture some elements of style. CodeT5 had the best performance for the classification task, so we next trained a set of sequence to sequence models to perform each style transfer. All of these experiments were based on CodeT5 small because the base version had marginally better performance but significantly more parameters making it a less viable candidate for the larger number of sequence to sequence models we trained. We set the baseline with CodeT5 architecture but being trained from scratch in order to evaluate how much CodeT5 can learn with or without pre-training code.

6.4.1 Individual Style Transfer

We fine-tuned CodeT5 for each of the parallel corpora we generated. All models were trained with the same hyperparameters. For docstring, comments, casing, and class corpora, each of the models was fine-tuned for 4 epochs. List comprehensions had only 36,000 examples and was thus fine-tuned for 6 epochs. Docstring was fine-tuned on a 400,000 example subset of the 2.5 million examples to reduce training time and make dataset size more comparable to other tasks.

6.4.2 Multiple Style Transfer

The combined model was trained with a subset of each of the parallel corpora. For docstring 250,000 examples were used, for casing 200,000 examples were used, for comments 350,000 examples were used, for class 350,000 examples were used, and for list comprehensions 36,000 examples were used. Each of the data samples had a natural language prompt appended to it indicating the transformation to be performed. The model samples batches proportionally to size of training data and each batch only contained examples from one of the parallel corpora. The model was trained for 12 epochs over this dataset with the same hyper parameters as the individual models. The model was tested every 4 epochs for performance on each of the individual tasks using the evaluation set.

In addition to the evaluation on individual tasks, the model was evaluated using the evaluation set on doing multiple transformations at once. The model was given a natural language prompt(figure 4 and figure 5), that specified that multiple transformations should be done and examples from the evaluation set where multiple transformations could be done were selected. Several combinations of features were performed with at most five features being done at once. All of these multiple transformations are zero shot tasks with idea being to test the model's ability to use what it learned in the individual tasks without the need to create different combinations of parallel corpora for fine-tuning multiple transformations.

7 Evaluation

We report our evaluation on multiple components: clustering to define code style, and style transfer using sequence to sequence generation.

7.1 Code Style Clustering

For clustering we evaluated the internal and external validity of the clusters.

7.1.1 Metrics

Davies-Boules Index For the internal metric, the Davies-Boules Index(DBI)(Davies and Bouldin, 1979) was selected; DBI measures the density of a cluster by calculating similarity across its containing data points. The score is considered better for the smaller value of DBI.

Purity For the external metric, we used Purity(Manning, 2008) to measure how well the clusters mapped to authorship of code scripts, with the assumption that a given author or a given organization will have the same style. Purity evaluates if a cluster can be recognized as a single class.

7.1.2 Qualitative Analysis

Hyperparameter Tuning For the results of the tuning experiments of HDBScan, we tuned three hyperparameters, which are minimum of each cluster size(Min Cluster Size), minimum samples for calculating the distance to the centroid(Min Samples), and Cluster Epsilon for the vague of the boundry between each cluster. The best setup gave the best performance on both purity and DBI. For Purity, the table 2 shows our clusterer performed a significant improve from random sampling baseline, which is randomly assign data points to clusters and calculate the purity with author labels.

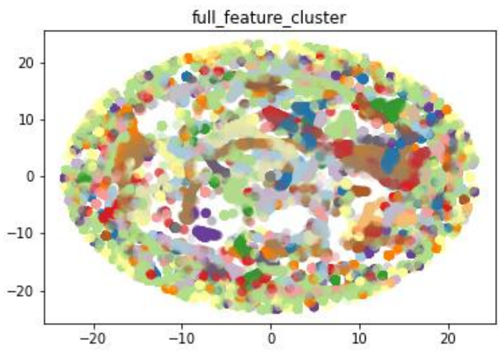


Figure 3: t-SNE(Van der Maaten and Hinton, 2008) visualization of the final clusters

Given the selected hyperparameters, we generated the final clusters from the full dataset. The figure 3 shows the t-SNE visualization of the data metrics with colors representing the cluster of the data point. The light green points indicate the data that were detected as outliers, which were around 741K examples. Other than those, there were around 234K examples being clustered out of 975K examples.

Pair Authorship Analysis Furthermore, we compared the pairwise authorship across all clusters for evaluating what the most distinctive coding style is when assuming each author represent a style. The score will only consider the data points that belongs to the pair of authors(authorship 1 and authorship 2). The improved score indicates how much it is improved from the random sampling baseline, which is randomly picking data points and assigning them to the clusters. The baseline score can be high simply just because of the high authorship distribution. So the improvement score is considered for targeting out the most valuable comparison by the HDBScan results. The table 3 shows the top comparing results of the evaluation. It suggests the *anhstudios* and *plotly* is having the most distinctive code style from others. The figure 6 shows an example of how their coding styles are different from each other.

Evaluation and Setup of our clusterer					
Model	Hyperparameters			Evaluation	
	Min Cluster Size	Min Samples	Cluster Epsilon	Purity	DBI
Baseline	-	-	-	0.246	-
Our clusterer	400	100	0.01	**0.404**	1.06

Table 2: The performance and setup for the final selected clusterer.

Cluster Authorship Pair Comparison					
Authorship 1	Authorship 2	Purity	DBI	Purity-Improve	DBI-Improve
anhstudios	plotly	**0.945**	0.845	**0.232**	**579.528**
anhstudios	AppScale	0.764	**0.838**	0.036	75.124
anhstudios	dropbox	0.884	1.125	0.029	88.904
anhstudios	mozilla	0.859	1.003	0.020	17.735
anhstudios	google	0.767	1.082	0.015	32.572
anhstudios	openstack	0.635	0.881	0.015	94.766

Table 3: The comparison on authorship pair for the quality of the clusters.

Performance on Traditional ML Classifier				
Classifier	Accuracy	Precision	Recall	F1
SVM	0.777	0.796	0.777	0.784
Logistic Regression	0.736	0.798	0.736	0.752
Decision Tree	0.980	0.980	0.980	0.980
Random Forest	**0.989**	**0.989**	**0.989**	**0.989**
Naive Bayes	0.755	0.571	0.755	0.650

Table 4: This is the evaluation for testing how well the quality of clusters is by fitting the cluster labels and data metrics with various traditional machine learning classifiers.

Traditional Machine Learning models We also evaluated the clusters using traditional classification algorithm with F-1 score, precision, recall, and accuracy. This evaluation tested the quality of clusters based on whether simple classifiers can map data points based on the same features we used for clustering into proposed clusters. The table 4 shows the performance score of measuring the predictability of the cluster for five different classifiers. The Naive Bayes was not performing well, randomly guessing the data points to the clusters. The logistic regression and SVM classifier were doing relatively better but still failed recognize many of cases. However, the tree-based classifiers were performing extremely well, especially for random forest classifier which reached almost 99% score for each metric. The reason is depending on how the HDBScan splitting the data points by using hierarchical tree structure. The results are indicating that our clustering results perform a good predictability on simple classifiers, but how will neural-based models do even without directly looking at the data features that the clusters are based on?

Pre-trained models Cluster Classification To ensure that pretrained models were capable of identifying aspects of code style, we used F-1, precision, recall, and accuracy to evaluate if the pre-trained code language models, PLBart and CodeT5, could classify the clusters based on purely on an input of source code. The table 5 shows the classification performance of different checkpoints on these two architectures. CodeT5 performed a lot more better results than PLBART on classifying the style clusters. The small size checkpoint and the base checkpoint only had very marginal difference on their scores, which means the compact version of CodeT5 was enough to capture the code style we defined with clusters. The reason of the better scores could contribute to the pre-training methods of CodeT5 which is more targeting on token-level

Performance for Pre-trained Code Language Model				
Model	Accuracy	Precision	Recall	F1
PLBart-Base	0.8	0.8	0.8	0.8
PLBart-Multi-task-Python	0.79	0.79	0.79	0.79
CodeT5-Small	0.919	**0.92**	**0.92**	**0.92**
CodeT5-Base	**0.92**	**0.92**	**0.92**	**0.92**

Table 5: The classification report for the pretrained language model finetuned on cluster classification.

prediction, especially for identifiers. Identifiers defined by users contain a lot more style information about the code than other keyword tokens. It also has the pre-training task of bi-modal dual generation, which will have understanding of the alignment between the code and the documentation. Overall, CodeT5 has capability of capturing the code style according to our proposed clusters even without seeing style features. But will CodeT5 be able to perform style transfer?

7.2 Code style transfer using Seq2Seq Generation

We used the following metrics to evaluate sequence to sequence generation.

7.2.1 Metrics

CodeBLEU The sequence-to-sequence generation task is evaluated with CodeBLEU(Ren et al., 2020), which is considered better than BLEU for code because it also compares code syntax and semantics. It leverages keywords of programming languages, the abstract syntax tree structure, and the semantics of programs. The final CodeBLEU score is an average value over BLEU score (N-gram), weighted BLEU score (Weighted N-gram), the syntax score (Syntax Match), and the semantic score (Dataflow Match).

BLEU For natural language tasks such as docstring transfer and comment transfer it is more appropriate to evaluate them with the standard BLEU score(Papineni et al., 2002).

DiffBLEU Since the CodeBLEU is too optimistic for evaluating the transfer task, so we proposed DiffBLEU, which is BLEU only on the difference between reference code X and prediction code $\hat{Y}$ and difference between input code X and label code Y. CodeBLEU sometimes shows a high score which just simply contribute to the successfully reconstructed code from input and other than the part that should be transfer. Especially for list comprehension transfer, the portion that need to be modified by the model is significantly small and the CodeBLEU can still show high performance even if it is not attempting any transfer. DiffBLEU is capable of evaluating only the difference of the code that should be perform in the transfer task and is applicable on all transfer tasks.

$$DiffBLEU = BLEU(Diff(X, \hat{Y}), Diff(X, Y)) \quad (1)$$

The equation 1 shows the concept of the DiffBLEU calculation. The method of extracting difference, *Diff*, is using the “SequenceMatcher” function from the built-in library “difflib”. However, the DiffBLEU is not perfectly precise on capturing the difference when multiple positions and large portion of change happened. DiffBLEU can only serve as a proxy for evaluating relative performance on transfer part and will need a more precise metric such as parsed-tree-based difference score.

Parsability We have also evaluated the validity of the generated code with the parseability metric. Parseability is the accuracy of whether the code is still parseable from the AST module in Python3. Any error occurrence during parsing will be considered as failing to parse.

7.2.2 Individual Style Transfer

According to the table 6, the baseline was reasonably performing scores closed to zero since it is hard to transfer style without any pre-trained understanding about the programming language. The scores show that the almost all individual finetuned models were performing the best for both maintaining generated code quality and performing transfer tasks.(See figure 7 for example.) The combined finetuned model were trained with multi-task learning, which was giving noises affecting the individual transfer results.

However, for the docstring transfer, the combined model got better performance on CodeBLEU.

Overall Performance for Seq2Seq Generation on Individual Style Transfer				
Task/Model	CodeBLEU	BLEU-NL	DiffBLEU	Parsability
Comment Transfer				
Baseline	0.031	0	2.20E-82	0.000
Individual Finetuned	**0.792**	**0.14**	**0.283**	**0.91**
Combined Finetuned	0.779	0.049	0.23	0.896
Class Transfer				
Baseline	0.028	-	1.52E-234	0.000
Individual Finetuned	**0.848**	-	**0.45**	**0.98**
Combined Finetuned	0.807	-	0.37	0.918
Docstring Transfer				
Baseline	0.027	0	3.43E-234	0.000
Individual Finetuned	0.656	0	**0.008**	**0.847**
Combined Finetuned	**0.715**	0	0.002	0.843
Casing Transfer				
Baseline	0.026	-	2.53E-235	0.001
Individual Finetuned	**0.967**	-	**0.728**	**0.947**
Combined Finetuned	0.894	-	0.382	0.925
List Comprehension Transfer				
Baseline	0.021	-	0.000	0.000
Individual Finetuned	**0.982**	-	**0.862**	**0.869**
Combined Finetuned	0.924	-	0.499	0.793

Table 6: The overall performance table on individual feature transfer.

One of the analysis is the combined model learned from a greater variety of training data than just the parallel corpus only on docstring transfer. So its ability of code reconstruction can perform on a larger distribution of code compared to the individual model.

Manual Evaluation on Docstring Transfer Manual evaluation was necessary for some natural language generation tasks such as docstring/comments transfer. Automatic scores for the natural language can mislead the true performance by giving low score without considering the meaning of the generated text and the sensibility of the model. So we evaluated the docstring transfer with manual evaluation on two metrics: **Sensible** and **Meaning**.

Meaning metric was a 1 to 3 scale depending on how well the meaning of the predicted docstrings match to the reference docstrings. The table 9 shows the description of each scale. If a docstring did not have a corresponding docstring in the ground truth, it was considered as not having a similar meaning.

Sensibility was measured on a 0 or 1 basis to see if the docstrings produced for each code snippet are grammatical and are not logical nonsense.

The manual evaluation(table 7) indicates that the model are capable of generating docstring by sensing the contents of the code. However, the produced meanings are too simple and not matching well to the targets.

7.2.3 Multiple Style Transfer

According to the table 8, the combined model was performing all significantly better than the baseline. However, the scores are performing in a lot more worse range comparing to the individual transfer. Even though the model can still generate the grammatical output, a lot of predictions are same as input, not performing any transfer. Some of them were only performing a subset of the transfer instead of the full combined transfer. However, there were also some successful cases(see figure 8) which full combined transfer was attempted. This indicates the CodeT5-small revealed the capability of multiple code style transfer at once.

8 Conclusion

In this work, we performed a novel task - code style transfer with individual and multiple style. We defined code styles through clusterings and found

Manual Evaluation on Docstring Transfer				
	Sensible		Meaning	
Task/Model	Avg.	Agreement	Avg.	Agreement
Docstring Transfer				
Individual Finetuned	0.848	0.848	1.283	0.435

Table 7: The manual evaluation on sensible and meaning of docstrings. The agreement score represent the percentage of how two annotators agree to each other.

Overall Performance for Seq2Seq Generation on Multiple Style Transfer					
Task/Model	CodeBLEU	BLEU-Comment	BLEU-Docstring	DiffBLEU	Parsability
Comment + Docstring					
Baseline	0.035	0	0	4.53E-82	0
Combined Finetuned	**0.560**	**0.003**	**0.005**	**0.003**	**0.839**
Casing + Class					
Baseline	0.026	-	-	1.51E-234	0
Combined Finetuned	**0.669**	-	-	**0.097**	**0.894**
List Comp + Casing + Class					
Baseline	0.019	-	-	0	0
Combined Finetuned	**0.667**	-	-	**0.065**	**0.827**
List Comp + Casing + Class + Docstring + Comment					
Baseline	0.025	0	0	1.84E-157	0
Combined Finetuned	**0.418**	**0.022**	**0.002**	**0.002**	**0.929**

Table 8: The overall performance table on multiple feature transfer.

the style clusters reached decent quality and predictability. We explored pre-trained code language models, CodeT5 and PLBART, and found CodeT5 has the strong capability to capture code style. We generated parallel corpora for our proposed tasks and fine-tuned CodeT5. The results strongly suggest that such a model can successfully learn the notion of code style and perform multiple style transfers at once.

Future Work For the further experiments, GAN training with combined finetuned model as the generator will be an interesting test. And we can also explore more powerful pre-trained language models such as BLOOM[1] and Codex(Chen et al., 2021) with prompting to examine whether and how they recognize the code styles.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *CoRR*, abs/2103.06333.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

David L Davies and Donald W Bouldin. 1979. A cluster separation measure. *IEEE transactions on pattern analysis and machine intelligence*, (2):224–227.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages.

[1] https://huggingface.co/bigscience/bloom

Di Jin, Zhijing Jin, Zhiting Hu, Olga Vechtomova, and Rada Mihalcea. 2020. Deep learning for text style transfer: A survey.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR.

Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *Journal of machine learning research*, 9(11).

Claudia Malzer and Marcus Baum. 2020. A hybrid approach to hierarchical density-based cluster selection. In *2020 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, pages 223–228. IEEE.

Christopher D Manning. 2008. *Introduction to information retrieval*. Syngress Publishing,.

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

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *SIGPLAN Not.*, 51(10):731–747.

Emily Reif, Daphne Ippolito, Ann Yuan, Andy Coenen, Chris Callison-Burch, and Jason Wei. 2021. A recipe for arbitrary text style transfer with large language models.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Parker Riley, Noah Constant, Mandy Guo, Girish Kumar, David Uthus, and Zarana Parekh. 2021. Text{settr}: Label-free text style extraction and tunable targeted restyling.

Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2022. Code translation with compiler representations.

Alexey Tikhonov, Viacheslav Shibaev, Aleksander Nagaev, Aigul Nugmanova, and Ivan P. Yamshchikov. 2019. Style transfer for texts: Retrain, report errors, compare with rewrites. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Association for Computational Linguistics.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation.

Mingzhe Hu Yun Peng, Yu Zhang. 2021. An empirical study for common language features used in python projects.

```
<nl> change for loop to list comprehension </nl><s>
from six import string_types
import re
import hashlib
from dolfin.compilemodules.compilemodule import com
```

Figure 4: Example individual transfer prompt

```
<nl> add docstring, list comprehension, class, casing, comment </nl><s

def init(bot):
    'Init.'
    bot = bot
    settings = nested_dict()
    settings.update(dataio.load_json(json))
    _config = None
```

Figure 5: Example multi-transfer prompt

```
from __future__ import absolute_import
from plotly.utils import _list_repr_elided


class InputDeviceState:
    def __init__(
        self, ctrl=None, alt=None, shift=None,
    ):

        self._ctrl = ctrl
        self._alt = alt
        self._meta = meta
        self._shift = shift
        self._button = button
        self._buttons = buttons

    def __repr__(self):
        return """\
```

Org: Plotly
Code for Visualization Tool
OOP-like

```
helmet.stfName('wearables_name', 'armor_mandalorian_helmet')
helmet.condition = 0
helmet.max_condition = 1000
inventory.add(actor, helmet)


belt.stfName('wearables_name', 'armor_mandalorian_helmet')
belt.condition = 0
belt.max_condition = 1000
inventory.add(actor, belt)

bicep_l.stfName('wearables_name', 'armor_mandalorian_bicep_l')
bicep_l.condition = 0
bicep_l.max_condition = 1000
inventory.add(actor, bicep_l)
```

Org: ANH Studios
Code for Game Server
Script-like

Figure 6: Example of a pair of scripts with distinctive code style: the code from *plotly*, which is used as the component for visualization tools, tends to be object-oriented designed; on the other hand, the code from *anhstudio*, which is used for game server, is having the style of single execution script. They are having difference on class, casing, and docstring usages.

Meaning Scale Table	
Scale	Description
1	No predicted docstrings have any similar meaning to label docstrings.
2	Some predicted docstrings have similar meaning to labels.
3	More than half predicted docstrings have similar meaning to labels.

Table 9: Scale Table for Meaning Metric

```
import random
import unittest
from algo.charseq import StrView

class TestStrView(unittest.TestCase):

    def test_perf(self):
        n = 100000
        s = ''.join(map(chr, [random.randint(ord('a'), ord('z')) for _ in range(n)]))
        suffixes = [StrView(s, i, (n - i)) for i in range(n)]
        for i in range(n):
            suffixes.append(StrView(s, i, (n - i)))
        suffixes.sort()

    def test_len(self):
        s = 'hello'
        self.assertEqual(len(StrView(s, 0, 0)), 0)
        self.assertEqual(len(StrView(s, 0, 1)), 1)
        self.assertEqual(len(StrView(s, 0, 2)), 2)
```

Figure 7: List comprehension transfer examples: the red text is the original text and the green text is the changed part.

```
'\n
class RepresentationGenerator(RepresentationGenerator):
    """
    The standard word-level format: 3 files, source, target, tags, one line per file, whitespac
e tokenized\n    '

def init(sourcefile, targetfile, tagsfile, return
    """

    def __init__(self, source_file, target_file, tags_file, return_files=True):
    data = parse_files(source_file, target_file, tags_file, return_files=return_files)

    self.data = parse_files(source_file, target_file, tags_file, return_files=return_files)

    @staticmethod
def parsefiles(sourcefile, targetfile, tagsfile, return    def parse_files(source_file, target_
file, tags_file, return_files=True):
        with codecs.open(source_file, encoding='utf8') as source:
            source_lines = [line.split() for line in source]
        for line in source:
            source_lines.append(line.split())
```

Figure 8: List comprehension + Casing + Class transfer examples: the red text is the original text and the green text is the changed part.