CodeSage:
A Generative Approach to Improving Code Quality

Shounak Ray, Michael Nath, Joseph Tey
Department of Computer Science
Stanford University
[shounak, mnath, joetey]@stanford.edu

Abstract

We propose a novel system, CodeSage: a generalizable method to generate higher-quality code snippets. While most studies focus on specialized models for code quality improvement tasks, such as fixing bugs, the uniqueness of CodeSage lies in using a Github reputation score to evaluate the quality of our functions. We break our task into three main contributions: 1) aggregate clusters of code functions where each cluster represents one specific intent category; 2) within each cluster, score the quality of each function using a unique Github reputation score and designate the top ones as high quality code snippets of that intent category; 3) aggregate a dataset mapping low-quality functions to corresponding high-quality; and 4) fine-tune a pre-trained Seq2Seq transformer model that generates a high-quality code function given low-quality code. Our main contributions include offering a generalizable and dynamic way to identify "intent" behind code snippets, publishing a dataset for the public annotated with reputation scores, and confirming robustness of our final Seq-2-Seq model through comparative analysis with silhouette scores.

1 Introduction

Often times, there is a more preferable and efficient implementation of the code a developer has written for their project. It is only through code reviews and extensive research that one is able to enhance their code quality, both from a functionality and style standpoint. It is especially a less fluid and time-consuming process for fledgling coders to gain inspiration from *relevant* and *exemplary* repositories online. Thus, the overarching problem we wish to address is the following: how can we provide developers with "better" alternative code snippets, and explain why such alternatives are better?

Most studies in the past focus on providing developers with 'similar' alternative code snippets (as opposed to higher quality), and if they do focus on producing 'better' code snippets, such models often focus on a specific, specialized definition of 'better', such as the specific task of repairing broken code. This paper is an exploratory study that proposes CodeSage: a novel, generative method of transforming 'bad code' to 'good code', that is both scalable and generalizable.

Rather than constraining the solution to specific types of 'bad code' and 'good code', CodeSage relies on a unique Github reputation score (derived from a combination of stars, forks and watchers) to infer code quality - a generalizable, data-driven approach to improving the quality of one's code.

This method enables the possibilities of many useful downstream tasks, although explorations of such tasks are beyond the scope of this paper. These tasks include using the produced 'good code' to find similar alternative snippets in large-scale databases like Github, or produce reasoning explaining how specifically a developer should improve their code.

2 Related Work

There exists a range of code retrieval systems that aim to find similar code snippets from a large-scale database. CodeHow Lv et al. (2015) uses NLP queries and code search techniques to find relevant snippets, and DeepCode Gu et al. (2018) leverages uniquely constructed code embeddings to also

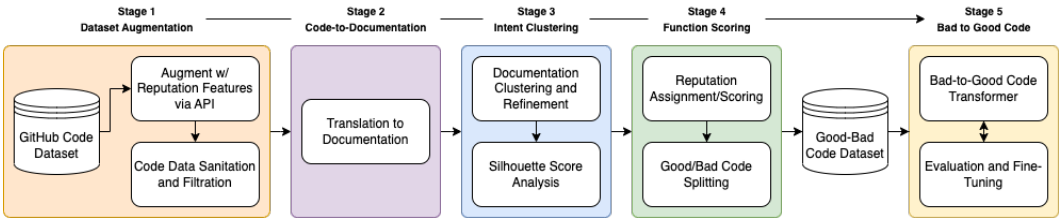

Figure 1: This figures depicts the pipeline used to generate “good" code from “bad" code. Starting at the left-most component, we ingest the `GitHub Code Dataset`, augment each constituent function with reputation features, and constrain ourselves to functions under a certain maximum length. In stage 2, we further augment function with a documentation string outputted by an LLM. In stage 3, a clustering algorithm intakes documentation for all code snippets and groups them into various “intent" clusters. For each identified cluster, Stage 4 assigns a reputation score to every constituent function per the calculated metrics in Stage 1. This is then used to strategically split each cluster into “good" and “bad" code snippets to construct a dataset, which is used to finetune a pre-trained code-to-code transformer in Stage 5.

recommend similar code snippets. However, when a developer is looking to improve the quality of his code base, it is likely that they are seeking 'higher-quality' versions of their code, as opposed to 'similar' code functions.

With regard to the systems that seek to improve the quality of code, such research is often focused on specialized tasks, including code correction tasks. Neural Progam Repair studies, such as the works of Tufano et al. (2019) and Huang et al. (2021), describe novel Seq2Seq models that suggest more reliable code snippets to replace faulty ones. Other specialized code quality improvement systems include Graph2Diff Tarlow et al. (2020), using graph-based representations of historical code changes to provide recommendations that specifically fix build errors. While there is a rich body of models that specialize in a range of code quality improvement tasks, such models are not designed to generalize beyond their area of focus.

The challenge of building a generalizable code quality improvement model is finding an appropriate dataset and code quality measurement that is also generalizable. Microsoft's CodeReviewer Li et al. (2022) is a model trained on real-world code reviews, and is able to generalize given the nature of such a dataset. CodeSage proposes an alternative to creating a generalizable code quality improvement model by using GitHub reputation scores in order to determine 'code quality', inferring characteristics of 'good code' and 'bad code' through such 'social' features.

3 Approach

We deconstruct this section into three main discussions: one outlining how we construct our data set, one outlining our baseline architectures and another outlining – in greater specificity – our custom data architecture. Both discussions include high-level justifications behind the design decisions for the respective architectures presented. See **Figure 1** for an overview of our general approach.

3.1 Data Augmentation

For our base dataset $\mathcal{D}$, we first leverage the `GitHub Code`[1] dataset available on HuggingFace, where each source code is broken down into its constituent functions - we ended up with 8,681 unique Python functions. Formally, $\mathcal{D} = \{x^{(j)}\}_{j=1}^{N_\mathcal{D}}$ of $N_\mathcal{D}$ functions from GitHub repositories, where $x^{(j)}$ consists of the source code c_j of the jth function. We update $x^{(j)} = x^{(j)} \cup \{f^{(j)}, s^{(j)}, w^{(j)}, p^{(j)}, d^{(j)}\}$ where – for the jth function – $f^{(j)}, s^{(j)}, w^{(j)}, p^{(j)}$ denotes the number of forks, stars, watchers, and issues of the repository and $d^{(j)}$ denotes a natural language description. For each code function $x^{(j)} \in \mathcal{D}$, we compute its 'reputation score' as $r_j = R(f_j, s_j, w_j, p_j)$, where $R : \mathbb{R}^4 \to \mathbb{R}$ denotes a scoring function. We chose a quadratic function that is as follows: $R(f_j, s_j, w_j, p_j) = s_j^2 + f_j^{1.5} + w_j + p_j$. This function attaches more weight to the number of forks and stars a repository has, with the motivation that variations in these features are more prevalent and meaningful than in the number of watchers and open issues.

[1] `https://huggingface.co/datasets/codeparrot/github-code`

3.2 Baseline Architecture – Two Avenues

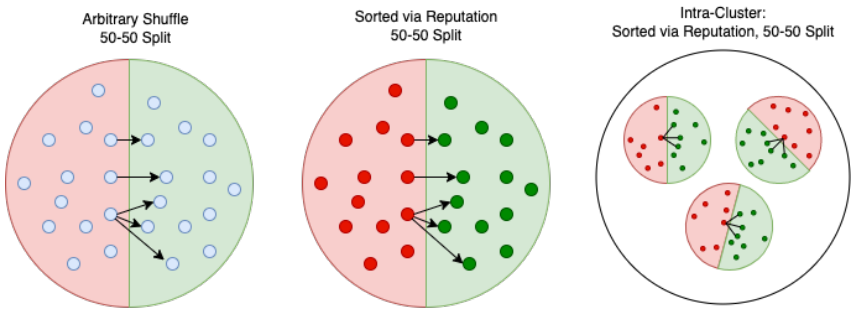

Figure 2: This figure shows the three different methods to segregate good and bad code considered to solve the greater objective of *generating* good code from bad code. The smallest circles represent a code snippet including its associated documentation. The arrows connecting the red-area circles to green-area circles represents the pairwise construction of our dataset for our final code-to-code model – as mentioned earlier. The left and middle arrangements represent the two baseline avenues discussed below, where the former involves randomly assigning half the functions to be bad code. The latter involves ranking based on a reputation score from least to greatest and assigning the first half to be bad code. The right-most figure is an outline of the CodeSage architecture, where code snippets are first clustered as per their documentation, subsequently assigned a reputation score, and then split in half

There are two baseline architectures which we will compare our novel `CodeSage` implementation to. The naïvest of these two begins by retrieving the our dataset $\mathcal{D}$, randomly shuffling the functions into an arbitrary ordering, and finally assigning the first half of these functions as "bad" code and the second half as "good code." The – anticipated – slightly more robust implementation begins by assigning a reputation score to each function according to a function outlined in **4.6**, ordering the functions according to this score, and then following the identical process of assigning the first half as bad code and the second half as good code. We hope these distinct baselines aid in elucidating the impact of supplementing a reputation score for each snippet.

Given these three implementations (two baselines and our custom approach), we are able to investigate three questions to assess different components of our custom approach. First, when comparing the baseline arbitrary to the baseline reputation-sorted approach, we will be able to discover whether fixing a reputation threshold to differentiate between code and bad code will help at all. Secondly, when observing the performance of our custom approach against the baseline reputation-sorted approach, we can identify clustering code by intent was beneficial to the final bad-to-good code generation. Finally, comparing performances of the naïvest and custom approaches, we will be able to discover how much more helpful clustering and reputation-sorting was towards reaching our objective.

3.3 CodeSage Architecture – Custom Implementation

3.3.1 Stage 1 – Documentation generation (Code-to-Documentation)

The first step is to extract the intention (the 'purpose' of a code block) of each function in our base data set, so that we can generate unique clusters corresponding to different intentions. We experimented with various ways of generating natural language text descriptions $d^{(j)}$ of each code function $f^{(j)}$. We tested two main approaches to extract function intent.

Firstly, we tested a `CodeTrans` model Elnaggar et al. (2021) specifically for generating function documentation for Python code, fine-tuned on the `CodeSearchNet`[2] dataset. While there were `small, base, and large` variants of this model, we opted to use the `CodeTrans t5-base` model as it achieved the highest `BLEU` score for Python function documentation generation (20.39).

Secondly, we also tested a `Davinci GPT-3` model to generate function documentation for Python code. We decided to use an LLM to generate documentation as we hoped that it would produce more detailed, granular descriptions of our functions. The configuration of the model was `temperature=0, max_tokens=50, top_p=1.0, frequency_penalty=0.0, presence_penalty=0.0`. We have

[2] `https://github.blog/2019-09-26-introducing-the-codesearchnet-challenge/`

been interested in investigating whether the granularity of our function documentation would impact the construction of our clusters that is done in the following stage.

3.3.2 Stage 2 – Intent Clustering (Document-to-Intent)

When provided with an arbitrary function provided by a programmer during inference, we want our code-to-code model to implicitly understand the intent of this input so as to accurately return similar-intent suggestions. To do so, the training set of the model must be constructed with respect to the intents of each function in $\mathcal{D}$. Let $\mathcal{I}$ denote the set of intent categories, which we have not discovered yet. Each function $x^{(j)} \in \mathcal{D}$ has a corresponding intent category $x_i \in \mathcal{I}$. We begin by vectorizing each function string $c_j \in \mathcal{D}$ through the `SentenceTransformer` model. We have developed two mechanisms to discover $|\mathcal{I}|$.

A brute-force approach is to test different k hyper-parameters fed into the `KMeans` clustering algorithm. This works by randomly selecting K initial cluster centroids and assigning each data point to the closest centroid based on a distance metric such as Euclidean distance. The mean of each cluster is then calculated and used to update the centroid location. This process is repeated iteratively until the centroids no longer move significantly, or a maximum number of iterations is reached.

The second clustering algorithm is Density-Based Spatial Clustering of Applications with Noise `DBSCAN`, an unsupervised method that ingests the vectorized function strings and outputs a fixed number of clusters (Ester et al. (1996)). The algorithm is hyper-parameterized by ϵ, which is the distance threshold for grouping points together. Points that are not within ϵ distance of any other point are considered noise.

3.3.3 Stage 3 – Function Scoring and Thresholding

Once we have clustered the functions based on similar intent, we proceed to scoring each one via the "reputation" of the GitHub repository it comes from. We perform this scoring to construct a heuristic that determines what is "bad" code, and what is "good" code. Our intuition is that while the number of stars, forks, open issues, etc of a parent repository suggest little about how poorly written a piece of code is, it can suggest something about how well it is written.

For each intent category $i \in \mathcal{I}$, after associating each function $x^{(j)}$ with a 'reputation score' r_j we then sort the constituent n_i functions in descending order according to the score. Given this sort, we now construct a set E_i of exemplar functions to which the set B_i of "bad" functions will be mapped to. At this point, we can implement a variety of thresholding functions that constructs this partition of our n_i functions into exemplars belonging to E_i, and non-exemplars belonging to B_i.

We denote a thresholding function as $T(r_1, r_2, \ldots, r_{n_i})$ that takes in the scores of the n_i functions in intent category i and outputs the 2-tuple (E_i, B_i). Note that it need not be the case that $|E_i| = |E_j|, |B_i| = |B_j|$ for intent categories i, j where $i \neq j$. Therefore, a meaningful thresholding function should be contextual with regards to the intent cluster it is operating on. To that end, we have explored two thresholding functions T_s and T_p. T_s represents the "boundary" thresholding function, which sends all functions having score at least $t \in \mathbb{R}$ to E_i, and otherwise to B_i. T_p represents the "percentile" thresholding function, which sends all functions whose scores are at a percentile greater than $p_h \in (0, 100)$ to E_i, and which sends all functions whose scores are at a percentile less than $p_l \coloneqq 100 - p_h$ to B_i.

3.3.4 Stage 4 – Bad to Good Code (Code-to-Code Model)

Let $N_{\mathcal{G}}$ denote the desired number of mappings we wish to finetune our code-to-code model on. We begin by constructing a new dataset $\mathcal{G}$ having $N_{\mathcal{G}}$ mappings by performing the following: within each intent category $i \in \mathcal{I}$, we generate new training instances by connecting each "bad" function in B_i to each exemplar function in E_i. Due to computational limitations, out of all the mappings, we randomly sample a maximum of 100,000 functions to yield $\mathcal{G}$.

Next, we proceed to fine-tune the `Salesforce CodeT5`[3] transformer on $\mathcal{G}$, training the model to map the input "bad" functions to the corresponding exemplar in $\mathcal{G}$. This transformer has been pretrained to handle a myriad of code generation tasks, some of which are exemplified in its open-sourced implementation available on Hugging Face (Wang et al. (2021)). Our motivation is that by observing

[3] `https://huggingface.co/Salesforce/codet5-base`

many translations of “bad" code into “good" code, the transformer may learn rich syntactic and semantic features of good code that can be used to improve an arbitrary piece of code.

Finetuning has involved experimenting with the learning rate α, the weight decay γ, and the number of finetuning epochs e. As a preprocessing step that conforms to the protocol of finetuning a T5 transformer, every ‘bad" code in $\mathcal{G}$ has been prefixed with the string “refine: ". This modified input, along with its good code label, are tokenized via the `RobertaTokenizerFast`[4] tokenizer available on Hugging Face. We finally replace each entry in $\mathcal{G}$ with the appropriate tokenized input and label.

4 Experiments

4.1 Investigating the Relationship between Document Generation and Cluster Performance

Given the interdependent nature of the `CodeSage` pipeline, it is possible that the hyper-parameter settings for one stage in the pipeline – say Stage 1 – could impact the performance of downstream stages – say Stage 2. In this section, we perform a silhouette analysis where we observe the impact of various Stage 2 code-to-documentation LLMs and Stage 3 clustering hyper-parameters on the effectiveness of the clustering algorithms. This analysis not only offers a point of comparison to the experimental results of `CodeSage` as presented in section 4.4 when evaluating the robustness of `CodeSage`, but also attempts to demystify the impact of clustering on downstream performance.

We vary the Stage 1 LLM used to generate documentation – such as `GPT` or `CodeTrans` – as well as the clustering hyper-parameters – including method (`KMeans` or `DBSCAN`), k-value (for `KMeans`), ϵ (for `DBSCAN`). To test the effectiveness of these aforementioned hyper-parameter combinations spanning stages 1 and 2, we compute the silhouette score S of the implemented clustering algorithm (Rousseeuw (1987). S is a useful metric to evaluate the quality of a clustering solution, as it measures the similarity of data points within a cluster and the dissimilarity of data points between clusters. A score of 1 means well-clustered data, -1 means misclassified, and a score close to 0 implies ambiguity between clusters. Note: “*detailed description*" refers to `GPT` in the figures below.

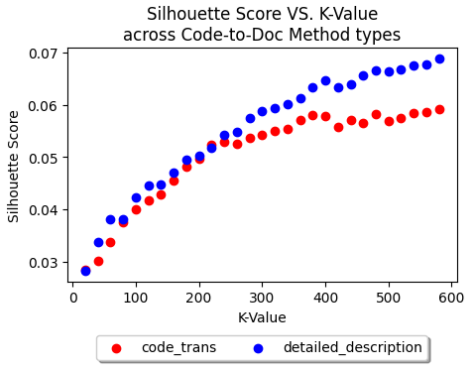

Figure 3: This figure shows how the silhouette scores varies across various k-values when the `KMeans` algorithm is used. We see that the utilities of both code-to-documentation algorithms is roughly the same until about $k = 250$.

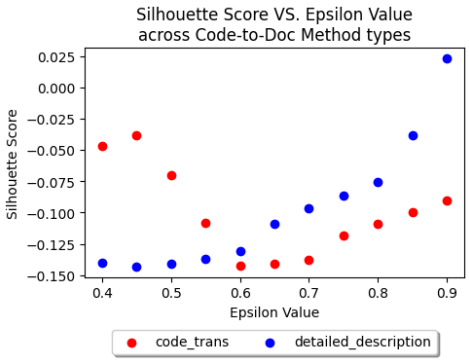

Figure 4: This figure shows how the silhouette scores varies across various ϵ-values when the `DBSCAN` algorithm is used. We see that the utility of the `CodeTrans` model is higher than that of `GPT` until about $\epsilon = 0.6$ when the order flips.

When the `KMeans` clustering algorithm is implemented, notice the divergence in S after $k = 250$ in **Figure 3**. This suggests that the `GPT` code-to-documentation model seems to be more performant and discriminatory than `CodeTrans` – however, only after $k = 250$. Since we’ve observed that $k \geq 250$ results in best performance – under the `KMeans` approach – we **first hypothesize** that the downstream code-to-code performance metrics will be superior in hyper-parameter combinations where $k \geq 250$ and `GPT` is used as opposed to combinations where $k \geq 250$ and `CodeTrans` being used. To test this hypothesis, we engineer downstream hyper-parameter combinations that incorporate k-values lower and higher than 250 across both the `GPT` and `CodeTrans` models.

[4] `https://huggingface.co/docs/transformers/v4.27.1/en/model_doc/roberta#transformers.RobertaTokenizerFast`

Considering **Figure 4**, we notice that $\epsilon \approx 0.6$ seems to be a turning point for which code-to-documentation approach yields the highest silhouette score. `CodeTrans` seems to have a higher silhouette score when $\epsilon < 0.6$ while `GPT` seems to have a higher silhouette score when $\epsilon \geq 0.6$. Therefore, we **secondly hypothesize** to observe higher code-to-code performance for `GPT` in hyper-parmeter combinations where where $\epsilon \geq 0.6$ and vice versa when $\epsilon < 0.6$. Notably, `DBSCAN` seems to – on average – have universally lower silhouette scores across all ϵ values compared to the `KMeans` approach for all k-values. This allows us to **finally hypothesize** that `KMeans` is generally a more superior algorithm to `DBSCAN`. We stress, though, that this is simply a hypothesis and there many other confounding hyper-parameter and data-modeling interactions that disagree with this notion.

4.2 Data

To reiterate, our primary dataset is the `GitHub Code` dataset found on Hugging Face. This dataset contains over 100 million code files spanning 32 programming languages - we filter specifically for `Python` functions. Unlike other datasets, this specific dataset is associated with a parent Github repository, allowing us to generate "reputation scores" that correspond to code quality.

4.3 Evaluation Method – Core Pipeline

Our evaluation method for the core pipeline involves computing a `CodeBLEU` and `CHRF` score of the predictions generated by our code-to-code model. While we considered computing the `BLEU` score, it is not tailored to the domain of code generation. The `CodeBLEU` and `CHRF` score have been reported to be a possible better alternative, considering the syntactic and semantic structure of code (Evtikhiev et al. (2022)). Before finetuning our pretrained code-to-code model, we create a validation dataset of proportion v of the total bad-code-to-good-code mappings in $\mathcal{G}$. After the finetuning epoch, we compute the three scores with respect to the input bad code in our validation split, the corresponding labeled good code, and the *predicted* good code output from our code-to-code model. Finally, we evaluate our code-to-code models on a test dataset of bad-code-to-code mappings that we have constructed from held-out entries of $\mathcal{D}$.

4.4 Experimental Details

Since the components of our system are interdependent, it is necessary to evaluate our system across various hyper-parameter configurations across each component. After taking into account the silhouette score analysis conducted in **4.1**, the memory limitations of our GPU, and the total virtual machine runtime available with our AWS credits, we have only investigated a subset of our hyper-parameters. Specifically, we performed a grid search over all hyper-parameters in our system, the realizations of which are described in **Table 3**, located in Appendix **A**. After finetuning is completed for each model, we deploy it on its respective test dataset (as described in **4.4**) and compute the average `CodeBLEU` and `CHRF` score. For finetuning, we have specifically finetuned the `Salesforce/codet5-small` Hugging Face model on an AWS g5.2xlarge EC2 instance. The test scores (500 mappings withheld) are enumerated in **Table 3** in Appendix **A**.

5 Results and Analysis

(a) Baseline versus CodeSage

Configuration	CodeBLEU	CHRF
Arbitrary Shuffle	25.255	5.458
Sorted by Reputation	25.226	5.514
CodeSage	**25.256**	**5.931**

(b) Test Scores Across Select Configurations

Code-To-Doc	ϵ or k	T	CodeBLEU	CHRF
CodeTrans	0.5	T_p	**25.257**	5.666
CodeTrans	0.85	T_p	2.439	4.417
GPT	0.5	T_p	2.399	4.414
GPT	0.85	T_p	25.256	**5.896**
CodeTrans	0.5	T_s	8.091	2.785
CodeTrans	400	T_s	5.133	4.931
GPT	60	T_s	0.331	5.277
GPT	400	T_s	5.058	5.701

Table 1: Results. Blue cells correspond to `DBSCAN` (where ϵ is a hyper-parameter), and green cells correspond to `KMeans` (where k is a hyper-parameter). For all of these configurations, the learning rate $\alpha = 0.001$, the percentile for $T_p = 90$, and the boundary for $T_s = 50$.

5.1 Baseline Versus CodeSage Results

Considering **Table 1 (a)**, we notice that CodeSage outputs code yielding both a greater CodeBLEU and CHRF score of code generated from our two baselines. We can now address the questions we have raised in **3.2**. First, it appears that mappings that consider the reputation scores of the involved functions induce a greater CHRF but lower CodeBLEU score than those induced by arbitrary mappings. Next, since CodeSage yields both a greater CodeBLEU and CHRF score than the Sorted by Reputation baseline, it suggests that intra-intent-cluster mappings induce code generations of higher quality. Lastly, comparing CodeSage against the Arbitrary Shuffle baseline, it suggests that the combination of intra-intnet-cluster mappings and reputation scoring yield the most performant generations with respect to the CodeBLEU and CHRF score. With that being said, the CodeSage architecture yields only marginally greater scores than the baselines. This suggests suggest that we may have to investigate with greater values of $N_{\mathcal{G}}$ and $N_{\mathcal{D}}$ to yield larger datasets that could yield more conclusive scores.

5.2 Grid Search Results and Analysis

Considering **Table 1(b)**, this is where our grid search results are shown. We have selected configurations that either yield the greatest scores, the lowest scores, or are simply worthwhile to examine. We notice immediately that with the experiments we were able to run, the CodeTrans model equipped with percentile-based thresholding and using DBSCAN clustering algorithm, where $\epsilon = 0.5$, achieves the best CodeBLEU of **25.257**. On the other hand, the GPT model equipped with percentile-based thresholding and using DBSCAN clustering algorithm, where $\epsilon = 0.85$, achieves the best CHRF scores of **5.896**, but has a lesser CodeBLEU score than at the analogous CodeBLEU setting. It is important to note that certain configurations yield relatively high CodeBLEU scores, but with low variance between them. We hypothesize that perhaps the code-to-code model overfits to a subset of the total mappings. Increasing $N_{\mathcal{G}}$ could alleviate this overfitting by yielding more variety in the training dataset, which downstream yields more variety in CodeBLEU scores among these configurations.

5.3 Hyper-parameter Analysis

Here, we will analyze notable results of the main hyper-parameters we tested, as shown in **Table 1 (b)**, and connect them with resultant hypotheses from the preliminary silhouette score experiments.

5.3.1 Code-To-Doc Model

Comparing rows 1 and 3 of **Table 1 (b)**, we observe that CodeTrans is better than GPT using the DBSCAN clustering algorithm, where $\epsilon = 0.5$, for both CHRF and CodeBLEU. Comparing rows 6 and 8 of **Table 1 (b)**, we also observe that GPT is better than DBSCAN using the KMeans clustering algorithm, where $k = 400$ – at least when considering CHRF.

In section 4.1, recall that our first hypothesis was that the code-to-code model would perform better on GPT than CodeTrans when $k \geq 250$. The results from our CodeSage hyper-parameter exploration presented in **Table 1b corroborates this first hypothesis**[5]. Specifically, this is the case when comparing row 8, which represents a KMeans configuration using GPT with $k = 400 \geq 250$ as opposed to row 6, which represents an identical configuration except with the CodeTrans model.

In section 4.1, recall that our second hypothesis was that GPT would be more performant than CodeTrans when $\epsilon \geq 0.6$, and vice versa when $\epsilon < 0.6$. The results from our CodeSage hyper-parameter exploration presented in **Table 1b corroborates this second hypothesis** – in terms of both CodeBLEU and CHRF.

The fact that both our first and second silhouette hypotheses were corroborated by the results from our hyper-parameter search suggests that CodeSage is robust – to an extent. Recall that the silhouette analysis was run on the entire, unfiltered function dataset $\mathcal{D}$ of 10,000 starting functions. Despite us only feeding the code-to-code transformer an upper-limit of 100,000 functions, the model was nevertheless able to *confirm* the expected relationships between the code-to-documentation LLMs, clustering methods, and clustering hyper-parameters. This offers an avenue to demystify the evidently positive impact of clustering on downstream code-to-code performance.

[5]Note that the CHRF scores a relatively wide apart. The CodeBLEU score only has a difference of one-tenth of a percentage point, which is arguably not enough to counter the evidence that GPT seems to be a superior code-to-documentation model than CodeTrans, at least for this KMeans hyper-parameter combination.

It is interesting to note that CodeTrans is exclusively superior to GPT under the DBSCAN clustering approach, whereas GPT is exclusively superior to CodeTrans under the KMeans clustering approach. One possible reason why is related to how the KMeans and DBSCAN algorithms function. KMeans is appropriate for well-defined clusters and roughly equal in size. That is, often when data has clear geometric structures and clusters are well-separated, KMeans may be a better choice than DBSCAN. The fairly $k = 400$ – for the configuration where GPT is better than CodeTrans – may imply clusters that are roughly equal in size given the forceful fragmentation into 400 distinct intent grouping, whereas DBSCAN – upon further analysis – outputs a mere 15 clusters. Therefore, higher-resolution intent grouping results in the construction of a superior dataset $\mathcal{G}$.

5.3.2 Clustering Algorithm

Comparing the overall blue and green configurations of **Table 1 (b)**, we also observe that DBSCAN performs on average, better than the KMeans clustering algorithm. These results, however, are **not consistent with the last hypothesis** of our preliminary clustering experiment, which predicted that KMeans should perform better DBSCAN. We suspect that this might be because of a few possible reasons. One possible reason is that DBSCAN is better suited than KMeans to handle noise and outliers in the original data, and this benefit is not accurately represented in the calculation of silhouette scores which calculates average distances between points. After all, it is plausible that there may have been a substantive number of "atypical" functions in $\mathcal{D}$ which lent DBSCAN to be a better algorithm. Secondly, there may be numerous other hyper-parameters in **Table 1b** at play – varying learning rates α and splitting methods T, for example, likely impact the *perceived* utility of DBSCAN. The impact of such downstream hyper-parameters were not captured by the upstream silhouette analysis – leading to such inconsistencies.

5.3.3 Thresholding Function

Comparing rows 1 and 5 of **Table 1 (b)**, this suggest that percentile-based thresholding functions (T_p) could be significantly better than boundary-based methods (T_s) when it comes to the quality of code generation. This could be because the boundary-based method, T_s, disregards the extent to which a score is below or above the selected boundary (in this case, 50). All functions that are sent to the same set, whether E_i or B_i, are treated as "equally bad" (or "equally good") which may not be appropriate. On the contrary, we can think of p_l and p_h parameters of T_p as controlling how discriminatory the thresholding should be in designating exemplars. For instance, if p_h approaches 100, then we are effectively filtering out possible exemplars to only the one with the maximal score. This may allow for more apparent bad-code-to-good-code mappings to be present in $\mathcal{G}$, which make it easier for the model to generate code like the exemplars.

6 Conclusion

CodeSage provides a novel consideration of code function documentation and GitHub features as a generalizable method for generating higher-quality code snippets. Firstly, we found that CodeSage performed better than both baselines, though this improvement was marginal. Further testing would be required to confirm statistical significance. Secondly, we found the GPT descriptions were better than CodeTrans descriptions when $k > 250$, and CodeTrans descriptions were better than GPT when $\epsilon < 0.5$. Thirdly, the main experimental results were consistent with silhouette analysis, suggesting that the accuracy of the clusters correlate with more accurate productions of higher-quality code. Finally, percentile-based methods of thresholding were better than shared methods, likely due to the former being more discriminatory. One of our key contributions include providing both detailed and general descriptions of over tens of thousands of arbitrary functions found on GitHub. We hope that this annotated dataset could be used in future work in the space of automatically improving code quality. Due to compute and memory constraints, we have been limited to finetuning a miniature version of the Salesforce CodeT5 transformer. We hope that leveraging larger versions, such as Salesforce/codet5-large may improve our code generation scores. We were also required to sample a maximum of 100,000 functions out of all the mappings from bad code to good code due to computational limitations. In the future, we predict that more data should lead more accurate results. Another major limitation was the large variety of different types of Python functions that may have limited the ability for meaningful clusters to be formed. Some functions were init functions for classes, while others were sorting and searching functions. Constructing a dataset that is more specialized for a specific function type may lead to more meaningful clusters, and therefore, more accurate results.

References

Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. 2021. Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing. *arXiv preprint arXiv:2104.02443*.

Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press.

Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2022. Out of the bleu: how should we assess quality of the code generation models?

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, pages 933–944.

Shan Huang, Xiao Zhou, and Sang Chin. 2021. Application of seq2seq models on code correction. *Frontiers in artificial intelligence*, 4:590215.

Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. 2022. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1035–1047.

Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 260–270. IEEE.

Peter J. Rousseeuw. 1987. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65.

Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. 2020. Learning to fix build errors with graph2diff neural networks. In *Proceedings of the IEEE/ACM 42nd international conference on software engineering workshops*, pages 19–20.

Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 25–36. IEEE.

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.

A Appendix

Table 2: Grid search hyper-parameter realizations

hyper-parameter	Possible Values
$N_{\mathcal{D}}$	{10,000}
$N_{\mathcal{G}}$	{100,000}
Code-to-Doc LLM Model	{`GPT`, `CodeTrans`}
Clustering Algorithm	{`DBSCAN`, `KMeans` }
Clustering Embedder	{`SentenceTransformer` }
ϵ	{0.5}
k	{30, 50, 60, 120, 180, 250, 400}
R	$\{R_q\}$
T	$\{T_s, T_p\}$
t	{50}
p_t	{60, 90}
α	{0.0001, 0.0003}
γ	{0.01}
e	{1}
v	{0.02}

Table 3: Test Scores Across Select Configurations

Code-to-Doc LLM Model	Clustering Algo.	ϵ	k	T	p_t	α	`CodeBLEU`	`CHRF`
`CodeTrans`	`DBSCAN`	0.5	-	T_p	90	0.0001	**25.570**	**7.165**
`CodeTrans`	`DBSCAN`	0.85	-	T_p	90	0.0001	2.439	4.817
`GPT`	`DBSCAN`	0.5	-	T_p	90	0.0001	2.399	4.714
`GPT`	`DBSCAN`	0.85	-	T_p	90	0.0001	25.140	4.461
`CodeTrans`	`DBSCAN`	0.5	-	T_s	-	0.0001	25.370	6.051
`CodeTrans`	`KMeans`	-	400	T_s	-	0.0001	5.133	4.931
`GPT`	`KMeans`	-	60	T_s	-	0.0003	25.224	5.587
`GPT`	`KMeans`	-	400	T_s	-	0.0001	5.058	5.701