Label Smoothing Improves Neural Source Code Summarization

Sakib Haque, Aakash Bansal, and Collin McMillan
Department of Computer Science
University of Notre Dame, Notre Dame, IN, USA
Email: {shaque, abansal1, cmc}@nd.edu

***Abstract*—Label smoothing is a regularization technique for neural networks. Normally neural models are trained to an output distribution that is a vector with a single 1 for the correct prediction, and 0 for all other elements. Label smoothing converts the correct prediction location to something slightly less than 1, then distributes the remainder to the other elements such that they are slightly greater than 0. A conceptual explanation behind label smoothing is that it helps prevent a neural model from becoming "overconfident" by forcing it to consider alternatives, even if only slightly. Label smoothing has been shown to help several areas of language generation, yet typically requires considerable tuning and testing to achieve the optimal results. This tuning and testing has not been reported for neural source code summarization – a growing research area in software engineering that seeks to generate natural language descriptions of source code behavior. In this paper, we demonstrate the effect of label smoothing on several baselines in neural code summarization, and conduct an experiment to find good parameters for label smoothing and make recommendations for its use.**

***Index Terms*—Source code summarization, automatic documentation generation, label smoothing, optimization**

I. Introduction

The backbone of much software documentation is the "source code summary" [1], [2], [3]. A summary is a short description in natural language that provides high level information about the purpose and behavior of the low level details implemented in source code. The idea is that a programmer can understand the functionality of a section of code by reading the corresponding summary without reading the code itself [4], [5]. The expense of writing these summaries by hand has long made automated code summarization a "holy grail" [6] of software engineering research [7]. The dream is that programmers could read code documentation even if other programmers did not write any.

The state-of-the-art in code summarization research depends on the neural encoder-decoder architecture [6], [8], [9]. Basically the idea is that an encoder forms a representation of source code in a vector space, while the decoder forms a representation of the summary in a different vector space. With sufficient training data (usually millions of samples [10], [11]), another part of the model learns to connect features in one space to the other and can be used to predict output summaries for arbitrary input source code. Neural designs based on the encoder-decoder model have almost completely supplanted earlier template- and heuristic-based approaches [7].

These neural models generate summaries one word at a time (usually using the teacher forcing procedure [12] in a seq2seq-like design [13], [7]). In a nutshell, the model is shown the source code and a start of sequence token for the desired output summary. The model then predicts the first word of the output summary. Then the model is shown the source code and the start of sequence token plus the first predicted word, then predicts the second word. This process continues until the model predicts an end of sequence token. The point is that the model is tasked with predicting a single word several times – not the whole output summary at once. Each individual word prediction is a weakpoint: if the model makes an error, the subsequent predictions are also likely to be wrong because they depend on the previous one [14]. If the model is "overconfident" in predicting some words, it may develop a tendency to miss rarer words and produce repetitive, dull outputs [15].

A solution proffered in several areas of natural language generation is *label smoothing* [16], [17], [18]. Label smoothing seeks to make the model "less confident" in each prediction by altering the target output. The neural model's output is a predicted probability distribution over the entire vocabulary of words. If the vocabulary has 10,000 words, then the output is technically a 10,000-length vector, in which the position of the correct word will hopefully have the highest value. During training, the target vector would have a 1 in the position of the correct word, and a 0 in the other 9,999 positions. What label smoothing does is reduce the value in the correct position slightly, say to 0.95, then spread the remainder over the rest of the distribution, so all other positions would be e.g., (0.05/9999). While the mechanism by which label smoothing works is not fully understood [17], different empirical studies have repeatedly shown it to be effective [16].

In this paper, we present an empirical study on label smoothing for neural source code summarization. Label smoothing requires considerable tuning to achieve optimal results in different domains. While it is reasonable to hypothesize that label smoothing would improve neural code summarization given the similarities between neural code summarization and other natural language generation technologies, the hypothesis has not been tested, and effective parameters and procedures have not been established. We conduct an experiment of label smoothing for several baselines from the source code summarization literature, to serve guide for future researchers.

II. Background & Related Work

In this section, we discuss the key background ideas and supporting work related to this paper, namely source code summarization, neural encoder-decoder architecture, and label smoothing.

A. Source Code Summarization

Early work in source code summarization included heuristic-based approaches [19], [20]. These models relied on techniques of Information Retrieval (IR) to extract salient words from source code [21], [22]. These words were then put into manually-defined templates to produce meaningful sentences [23], [24].

The landscape changed with the advent of deep learning. The explosion of data-driven models in the mid-2010's and their state-of-the-art performance in various natural language processing (NLP) tasks inspired researchers to use them to automatically generate source code summaries. Iyer *et al.* [25] was one of the earliest to use such models for code summarization. Since then, the field has embraced this sequence-to-sequence (seq2seq) architecture as the standard architecture to generate comments. Figure I lists some of the most prominent papers that use some modified version of an attentional neural encoder-decoder model to generate code summaries. This list is not exhaustive, but only includes peer-reviewed papers where a new idea was first introduced.

The table shows that the first generation of these data-driven approaches (marked only in column N) only looked at the source code tokens to generate comments. Later, Hu *et al.* [9] and LeClair *et al.* [6] noted the importance of including structural information about the source code by using the Abstract Syntax Tree (AST). They both flattened the AST into sequential tokens, but incorporated these tokens in different ways in their model. Other research papers soon explored various ways of capturing these structural information from the AST [29], [30], [35], [32]. At the same time, a parallel research track delved into incorporating contextual information. This context encompasses API calls to learn the mapping between API sequences and natural language description [28] as well as other functions in the file to provide supporting information for the code [33]. Bansal *et al.* further expanded the latter idea by including project context information [41]. Recently, some research has been dedicated into using transformer models [43] for this task. Ahmad *et al.* [36] used pairwise relationship between tokens to capture their mutual interaction beyond positional encoding while Gong *et al.* [42] introduces another layer in the encoder side of the transformer for the AST.

	N	S	C
Iyer *et al.* (2016) [25]	x		
Loyola *et al.* (2017) [26]	x		
Jiang *et al.* (2017) [27]	x		
Hu *et al.* (2018) [28]	x		
Hu *et al.* (2018) [9]	x	x	
Allamanis *et al.* (2018) [29]	x	x	
Alon *et al.* (2019) [30]	x	x	
Gao *et al.* (2019) [31]	x		
LeClair *et al.* (2019) [6]	x	x	
Fernandes *et al.* (2019) [32]	x	x	
Haque *et al.* (2020) [33]	x	x	x
Haldar *et al.* (2020) [34]	x	x	
LeClair *et al.* (2020) [35]	x	x	
Ahmad *et al.* (2020) [36]	x	x	
Zügner *et al.* (2021) [8]	x	x	
Liu *et al.* (2021) [37]	x	x	
LeClair *et al.* (2021) [38]	x	x	
Gao *et al.* (2021) [39]	x	x	
Wang *et al.* (2021) [40]	x	x	
Bansal *et al.* (2021) [41]	x	x	x
Gong *et al.* (2022) [42]	x	x	

TABLE I: Selection of closely-related, peer-reviewed works that use neural network based architecture. Column N indicates Neural Network inspired solutions. S indicates that structural data (various representations of AST) is used. C indicated publications that incorporate contextual information.

This paper aims to improve the performance of these data-driven models by exploring how label smoothing can enhance their performance. We show how the performance of different established baselines improves using label smoothing. Note that we do not seek to compete with any one code summarization approach – our aim is to benefit all approaches.

B. Neural Encoder-Decoder Architecture

The neural encoder-decoder architecture is the backbone of almost all current code summarization approaches [14]. These models are borrowed from the field of Neural Machine Translation (NMT). Essentially, these models have 2 parallel components: the encoder and the decoder. The encoder takes words (tokens) from the input language (e.g. English for NMT/Java for source code summarization) and represents them into fixed length vectors. The decoder takes this vector representation and translates them to the target language (e.g. Spanish for NMT/English for source code summarization).

The standard setup for these encoder-decoder models use recurrent layers on both sides [44]. These recurrent layers are typically a sequence of LSTM [45] or GRU [46] cells. Information propagates through these layers, one token per cell. Each cell in the encoder layer not only receives the corresponding token, but also the vector representation of all the tokens that came before. The first cell in decoder layer receives the final vector representation of the entire input sequence [13]. Consecutive cells in the decoder layer receive a vector that encapsulates not only the entire input sequence but also the output so far.

While this architecture has existed for some time, Bahdanau *et al.* [47] enhanced its performance by introducing attention in 2014. The intuition behind attention is that some tokens in the input sequence are more important than others when trying to generate specific predictions. The goal of attention is to map the relative importance of each input token to every output prediction. It does this by computing the similarity between the input sequence and the output sequence so far to identify the important features to predict the next word. The two most common attention functions are additive and multiplicative (dot-product based). In all the models evaluated in this paper, we use multiplicative attention.

The dot-product based attention inspired a new generation of neural encoder-decoder architecture called Transformers, that eschews the sequential nature of these models. First introduced by Vaswani *et al.* [43], these models replace the RNN cells with a self-attention layer (discussed in greater detail in section III-C) followed by a fully connected feed-forward layer. Without the recurrent layer to propagate sequential information, transformers introduce a positional encoding before the encoder and decoder respectively to capture the order of the sequence. This allows for parallel processing of tokens, making them faster. These models have also been shown to better capture long-range relationships [43].

C. Label Smoothing

Label smoothing is a regularization technique used in neural networks to improve generalization. It was first introduced by Szegedy *et al.* [48] for image classification, although it has since been shown to improve performance for many other deep learning tasks, including NMT [16]. Essentially, label smoothing involves introducing some uncertainty in the training data to prevent over-fitting [49], [17], [50].

Most Natural Language Generation (NLG) models use categorical cross-entropy loss to calculate the error between target tokens and predicted tokens. Minimizing this loss function means reducing the error between the target and predicted tokens, thus improving model performance. The target distribution ($t(k)$) for a neural network is a Dirac delta function:

$$t(k) = \delta_{k,y}$$

where k is the predicted output and y is the target output and $t(k)$ is 1 when $k = y$ and 0 for all other k. In practice, we use a one-hot vector to represent the Dirac delta function. The size of the vector is the number of tokens in the output vocabulary (N_t), with all elements set to 0 except for the index of the target token, which is set to 1. We can therefore read it as a probability distribution that is 1 for the correct target word and 0 for all other words in the vocabulary. This is undesirable because it makes the model overconfident about certain predictions.

With label smoothing, we encourage the model to be "less confident." We achieve this, in essence, by taking some probability, ϵ, off the top of the target token, y and uniformly distributing over the rest of the vocabulary. This probability, ϵ where $\epsilon \in (0, 1]$, is the key parameter that determines the extent to which label smoothing will affect model performance. We represent this new distribution as:

$$t(k) = (1 - \epsilon)\delta_{k,y} + (1 - \delta_{k,y})\frac{\epsilon}{N_t - 1}$$

where $t(k)$ is now $(1 - \epsilon)$ for $k = y$ and $\frac{\epsilon}{N_t - 1}$ for $k \neq y$.

The current research frontier in source code summarization lies in enhancing the neural networks to generate better comments. This paper lies at the heart of this frontier by not only quantifying the regularization effect of these models with label smoothing but also identifying the best configuration to maximize model performance.

III. Experimental Design

In this section, we discuss the design of our experiment. This includes the research objectives of this paper, the datasets we use to perform the experiments, the models we use to train the data, the metrics we use to evaluate model performance, and any threats to validity we might have.

A. Research Questions

Our research objective is to evaluate the extent to which label smoothing improves source code summarization models, and the configurations that maximize this performance improvement. To this end, we ask the following research questions:

RQ1 What is the effect of label smoothing on the overall performance of recently-published source code summarization baselines in terms of BLEU, METEOR and USE+c scores?

RQ2 What is the best label smoothing configuration that maximizes model performance and how is this configuration affected by the output vocabulary size?

RQ3 What is the effect of label smoothing on the diversity of target vocabulary and how does this affect performance?

The rationale behind RQ1 is that label smoothing is not a commonly used regularization technique in automatic source code summarization, despite evidence of success in other NLG applications. Many models have been proposed for source code summarization, but to the best of our knowledge, none use label smoothing as a regularizer. While there is evidence that label smoothing will improve the performance of these models, there has been no extensive research to demonstrate this. Our goal with this RQ is to quantify how including label smoothing to train these models can change their performance. We use three different metrics to verify our findings: BLEU, METEOR and USE+c. These metrics are further discussed in Section III-D

The rationale behind RQ2 is that our application of label smoothing has underlying hyperparameters that require tuning. As noted earlier, $\epsilon \in (0, 1]$. But the research literature is unclear about what the optimal values are for these hyperparameters, especially in the problem of code summarization. Additionally, we also do not know how the size of the output vocabulary (N_t) affects model performance with this configuration. The larger the vocabulary size, the smaller the smoothed probability per output token ($\frac{\epsilon}{N_t}$). Our goal with this RQ is to identify a suitable value of ϵ. Furthermore, we aim to understand the relationship of this probability ϵ with the size of output vocabulary, N_t.

The rationale behind RQ3 is that there are competing intuitions as to how label smoothing affects model performance. As noted in Section II-C, label smoothing artificially introduces uncertainty in the one-hot output vector. On one hand, the introduction of uniform uncertainty across the output vector could exacerbate the problem of label noise [51]. However, this uncertainty makes the model less confident about its predictions and allows it to consider rare words, thus

reducing the problem of class imbalance in source code summarization [14]. On the other hand, preventing overconfidence could mitigate label noise by improving model generalization, thus focusing on more commonly occurring words [17]. Our goal with this RQ is to study which of these competing intuitions is borne out in practice for the problem of source code summarization.

B. Datasets

We use two datasets in this paper: one is Java and the other is C/C++. The Java dataset, first published by LeClair *et al.*, introduce some of the best practices for developing a dataset for source code summarization that are now a standard among the research community [11]. It consists of 2.1m methods from more than 28k projects. The training, validation and test set are split by projects to prevent data from train set to leak into the test set by virtue of being in the same project. This dataset has since been used in many peer-reviewed publications [33], [35], [52], [41], [53] and new additions has since been made to it, including context tokenization. We use a filtered version of this dataset, with 1.9m functions, published by Bansal *et al.* that remove code clones in accordance with recommendations by Allamanis *et al.* [54].

The C/C++ dataset was first published by Haque *et al.* [14] following an extraction model proposed by Eberhart *et al.* [55] to adhere to the idiosyncrasies of C/C++, while maintaining the same strict standards proposed by LeClair *et al.* [11]. It consists of 1.1m methods from more than 33k projects.

Additionally, we extract individual statements on the top 10% largest methods from the Java dataset (Java-q90) and the top-25% largest methods from the C/C++ dataset (C/C++-q75). While this filtration reduces the size of the dataset, it eliminates simple getters/setters and other small functions whose comments are easy to predict. The remaining functions have more statements, which is more challenging and representative of real world use-case scenario. We chose top 10% for Java and 25% for C/C++ to keep the average size and number of subroutines similar for both datasets.

For RQ3, we use the method outlined by Haque *et al.* to convert both the Java-q90 and C/C++-q75 dataset to action word prediction dataset [14]. We extract action words from comments and stem them to reduce vocabulary size (e.g. to ensure that 'delete', 'deletes', 'deleted', 'deleting' are classified as the same action word).

C. Baselines

We use six baselines in this paper. We chose each baseline because it represents a family of similar approaches or is a well-cited approach used as a baseline in many papers.

attendgru This baseline is a simple unidirectional RNN-based attentional neural encoder-decoder architecture. It takes only source code tokens as encoder input and English comment as decoder input. It was first introduced by Iyer *et al.* [25] as an off-the-shelf NMT/NLG approach to generate source code summaries. For our implementation, we use GRUs instead of LSTMs because they are much faster while providing comparable performance [6].

transformer This baseline is another simple encoder-decoder architecture, but it replaces the recurrent layers with stacked muti-head attention layers [43]. As mentioned in Section II-B, transformers introduce a position embedding layer that captures the sequential order of tokens which allows the multi-head attention layer to process the entire sequence at the same time. On the encoder side, the multi-head attention layer computes dot-product based self-attention on the source code tokens. On the decoder side, there are two multi-head attention layers: a masked multi-head attention layer that computes self-attention on the comment tokens followed by a regular multi-head attention layer that computes attention between the encoder and the masked attention layer.

ast-attendgru This baseline is an enhancement over the attendgru model by including AST information on the encoder side along with source code tokens. This idea was first proposed by Hu *et al* [9] who designed a Structure-Based Traversal (SBT) algorithm to flatten the AST and include the source code and AST input together in the encoder. LeClair *et al.* improved this model by incorporating this flat AST on a separate recurrent layer and concatenating this encoder output with the original encoder output with just the code tokens [6]. For our evaluation, we use the implementation by LeClair *et al.* because it's more recent and performs better.

code2seq This baseline, proposed by Alon *et al.* [30] is similar to ast-attendgru as it also takes both source code tokens and AST as input. However, instead of flattening the AST, they encode pairwise paths between nodes in the AST. Then, they randomly select a subset of these paths as training input. Randomly selecting paths prevents over-fitting while keeping the model size reasonable. To reduce architectural variations and manage resource constraints, we set the number of paths explored to 100.

codegnngru This baseline provides another different technique for representing AST along the encoder. Proposed by LeClair *et al.*, it uses Convolutional Graph Neural Networks (ConvGNN) to process the AST input [35]. They pass the AST nodes through an embedding layer. The output of this layer along with the AST edge data is then input to the ConvGNN. For each node, ConvGNN adds the neighboring node to it's current node during a hop. The model shows best performance for 2 hops; each AST node adds the neighboring node twice, thus propagating information between nodes that are separated by 2 hops. The output of the ConvGNN layer is then passed to a recurrent layer before the result is concatenated with the output of the encoder that processes the source code tokens.

ast-attendgru-fc This baseline improves upon the ast-attendgru model by including file context information on the encoder side along with source code tokens and AST. Haque *et al.* applied the concept of using contextual information from other functions in the same file to a few different baselines [33]. They introduced a new encoder to process other functions in the file. All encoder outputs are combined before passing it to the decoder for prediction. While their results showed improvements in all baselines using file context, ast-attendgru-fc was the highest performing model.

D. Metrics

We evaluate these baselines using three different evaluation metrics: BLEU, METEOR and USE+c. Additionally, for RQ3, we use precision, recall and F1-score to evaluate action word prediction [56]. BLEU and METEOR are n-gram matching metrics while USE+c is a semantic similarity metric. We use BLEU because it is the most popular evaluation metric for source code summarization. BLEU is a precision based measure of N-gram overlap [57]. It was first introduced by Papineni *et al.* in 2002 for automatic evaluation of machine translation tasks. It compares n-grams in the predicted and target summaries. Typical implementations of BLEU scores set the range of n from 1 to 4. An average BLEU score is then computed by combining these individual n-gram scores using predetermined weights. Most code summarization models use a uniform weight distribution of 0.25 for each n, so we adhere to the same weights.

Recent studies [52] have shown that BLEU does not correlate well with human judgement of source code comments. Roy *et al.* [58] and Haque *et al.* [59] have proposed METEOR and USE+c as alternatives that better correlate with human evaluation. METEOR [60] was introduced in 2005 to address the concerns of using BLEU [57] or ROUGE [61]. It combines n-gram precision and n-gram recall by taking their harmonic mean to compute a measure of similarity. Like BLEU, METEOR tries to find exact n-gram matches. If however, an exact match is not to be found, it performs a second pass to match word stems and finally a third pass, to match word synonyms.METEOR does not have a long history of being used in source code summarization. However, it was recently shown to better reflect human quality assessment of generated summaries than BLEU or ROUGE, so we report this score.

USE+c [59] is a new evaluation metric proposed for source code summarization. It differs from BLEU and METEOR because it does not focus on n-gram matching. Instead, it uses the pre-trained Universal Sentence Encoder (USE) [62] to compute a vector representation of both target and predicted sentence. Note that USE trains stacked transformer encoders to compute context aware representation of words in a sentence before turning them into fixed length embeddings. USE+c then computes the cosine similarity between these fixed length vectors. A shortcoming of USE+c is that it does not include a brevity penalty. Unlike BLEU or METEOR, it is also difficult to explain. However, a recent survey by Haque *et al.* [59] showed that USE+c is better correlated with human ratings in terms of similarity, accuracy and completeness than most n-gram based metrics (including BLEU and METEOR). Therefore, we report USE+c as an additional metric.

E. Threats to Validity

Like any study, our experiment carries threats to validity. One key threat affecting this paper is the datasets we use. We aim to mitigate the first threat by using two peer-reviewed datasets with millions of examples in two different and widely-used programming languages. While we perform some additional filtration of the datasets, we still train our models on a couple hundred thousand functions. However, it is still possible the results may not be representative to all datasets. In this regard, we advise caution on generalizing our findings outside of Java and C/C++ datasets.

Another threat to validity is the plethora of implementation decisions we had to make for our baselines. Each baseline discussed in section III-C have different hyperparameters. These hyperparameters may change how label smoothing affect model performance. Each model is also different in terms of parameters tuned during training (model size). Due to hardware limitations, we do not compute the effect of label smoothing on each model using different model-intrinsic hyperparameters. Instead, we mitigate this threat by taking the best hyperparameters listed for each baseline. However, different hyperparameters could alter our conclusions.

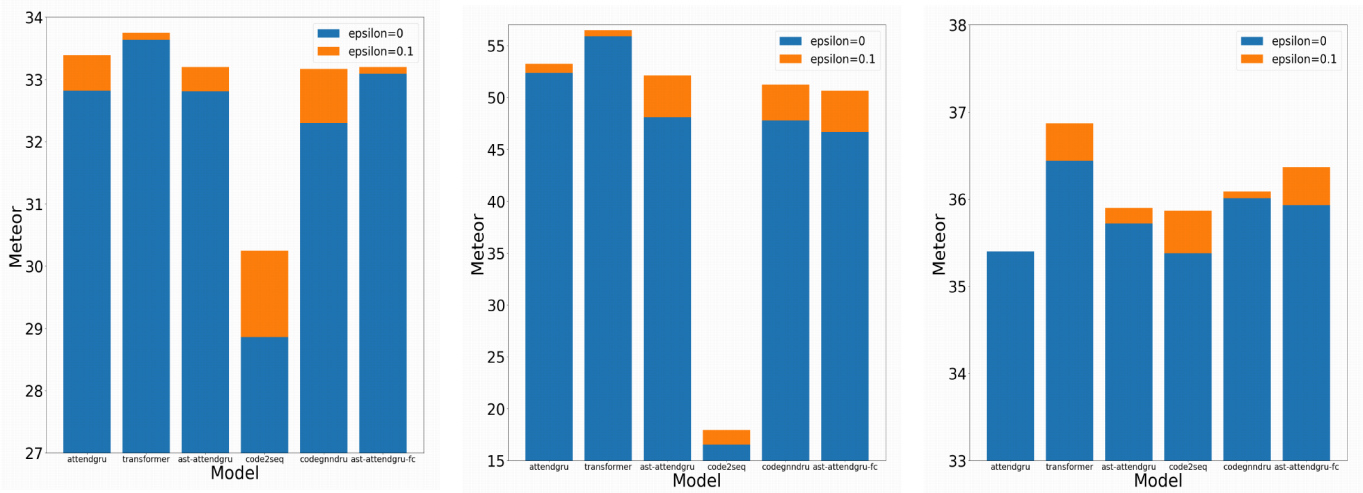


Fig. 1: Comparison of baselines with and without label smoothing (ϵ=0.10). Blue indicates baseline aggregate METEOR score. Orange indicates increase in METEOR score for identical model with label smoothing added. Note y-axis starts at 27 METEOR for the Java-q90 dataset (left), 15 for the C/C++-q75 dataset (center) and 33 for the full Java dataset figure (right).

IV. RQ1 - Overall Performance

This section discusses the methodology for answering RQ1 as well as our key findings. In general, we find that label smoothing leads to improvements to several baselines.

Methodology: To answer RQ1, we follow the established methodology that have become standard in source code summarization and NMT tasks in NLP [6], [43]. We use two standard datasets in two common programming languages. We perform further filtration on the dataset by taking the top 10% largest methods from the Java dataset (Java-q90) and the top-25% largest methods from the C/C++ dataset (C/C++-q75). We do this to find functions that have a large number of statements, which is more representative of real world use-case scenario. We also train the models on the full 1.9m Java dataset to evaluate model performance on a full dataset to make commensurable comparison with related work. For all three dataset instances, we train each baseline twice: once with and again without label smoothing. We keep the value of ϵ constant (0.1) for all models. For each architecture, we train both instances (with and without label smoothing) for 10 epochs and choose the model with the highest validation accuracy score for our comparison (standard practice in related work). We then evaluate the performance of these models using automated evaluation techniques discussed in Section III-D. Roy *et al.* at FSE'21 recommended to perform a paired t-test between baseline model predictions and new model predictions for different sentence-level metrics. Therefore, we perform a paired t-test to identify if the performance difference between two comparing model predictions is statistically significant. Note that we do not perform a t-test on BLEU score because BLEU is a corpus level metric.

Key Findings: Our key finding in answering RQ1 is that adding label smoothing as a regularizer improves model performance in most cases. All models show performance improvement in terms of USE for Java-q90, C/C++-q75 and full Java dataset. All models using the Java-q90 and C/C++-q75 datasets and most models using the full Java dataset also report performance improvement in term of METEOR and BLEU. Furthermore, most of the performance improvements are statistically significant (we choose $\alpha = 0.05$). This overall trend of performance improvement is depicted in Figure 1. The dark blue bar shows the METEOR scores for models without label smoothing. The orange bar on top shows the increase in the METEOR score after we add label smoothing. As the figures show, there is an increase in METEOR score in almost all cases.

Table II shows the effect of label smoothing on the overall performance of each baseline for the Java-q90 dataset. We find that all models report higher evaluation metric score with the addition of label smoothing. However, Code2seq improves the most with 4.8% improvement on METEOR, 5.4% improvement on USE and 8.3% improvement on BLEU. Codegnngru also shows significant improvement with 2.7% improvement on METEOR, 2.3% improvement on USE and 4.2% improvement on BLEU. Attendgru and ast-attendgru-fc also show similar improvement with 1.7% improvement for both on METEOR, 2.4% and 1.7% improvement on USE and 3.1% and 3.3% improvement on BLEU respectively. Each of these models vary with respect to encoder input. Each model has different inputs as well as different data structure for common inputs. The higher scores are encouraging as we observe statistically significant performance improvement across different model architectures. Transformers, however, show the least overall improvement in performance with an increment of less than 1% for all metrics, although the 0.85% increase in USE score is statistically significant. One possible explanation for this result is that Transformers have built-in dropout layers after each multi-head attention layer. These dropout layers likely already improve regularization.

Table III shows the effect of label smoothing on the overall performance of each baseline for the C/C++-q75 dataset. For this dataset, we notice a large improvement in performance for ast-attendgru, code2seq, codegnngru and ast-attendgru-fc baselines. These performance increase range from 7.2%-8.5% for METEOR, 5.6%-10.3% for USE and 8.3%-10.2% for BLEU. It is interesting to note that attendgru does not show a large percentage increase in performance ($<$2% across all metrics) for this dataset, compared to the aforementioned models. Furthermore, similar to the Java-q90 dataset, transformers also show little improvement (about 1-2% across all metrics). We expect this result for transformers due to the built-in dropout regularizer in the transformer architecture. However, the performance increase for both attendgru and transformers is still statistically significant with p-values $<$0.01 for both METEOR and USE.

Table IV shows the effect of label smoothing on the overall performance of each baseline for the full Java dataset of 1.9m methods. Again we see a trend of performance improvement in most cases. However, while the increase in metric score is significant in all but two configurations, we do notice that the percentage increment is not as high as for the Java-q90 and C/C++-q75 dataset ($<$2% across all metrics for all models). We attribute this to the fact that higher training examples act as a regularizer in and of itself. For attendgru, we notice a statistically significant decrease in METEOR score but a statistically significant increase in USE score. We also notice a small decrease in BLEU score. Since all three metrics do not point in one way or another, we do not draw any conclusion for this setup. For codegnngru, we notice a 0.11% decrease in BLEU score, a 0.22% increase in METEOR score which is not statistically significant but a 0.47% increase in USE score which is statistically significant. Since the two metric scores that show performance improvement are also correlated best with human evaluation, we conclude that label smoothing positively affects codegnngru. Once again, code2seq shows the highest performance increase for the Java full dataset. One likely explanation for why code2seq achieves the highest performance increase is that it is the largest model among the baselines. However, its parameters do require more time to train than others. Like for other models, label smoothing appears to help code2seq generalize.

TABLE II: Effect of label smoothing on Java q90 dataset ($\epsilon = 0.1$, $N_t = 10908$)

	without label smoothing			*with label smoothing*			*percentage difference*			*p-value*	
Model Name	METEOR	USE	BLEU	METEOR	USE	BLEU	METEOR	USE	BLEU	METEOR	USE
Attendgru	32.82	50.21	18.87	33.39	51.41	19.45	1.74%	2.39%	3.07%	<0.01	<0.01
Transformer	33.64	51.81	18.99	33.75	52.25	19.11	0.33%	0.85%	0.63%	0.49	<0.01
AstAttendgru	32.81	50.20	18.61	33.20	50.88	19.12	1.19%	1.35%	2.74%	0.02	<0.01
Code2Seq	28.86	43.99	14.95	30.25	46.37	16.19	4.82%	5.41%	8.29%	<0.01	<0.01
CodeGNNGRU	32.30	49.37	18.00	33.17	50.49	18.76	2.69%	2.27%	4.22%	<0.01	<0.01
AstAttendgru-fc	33.09	50.05	18.76	33.66	50.91	19.38	1.72%	1.72%	3.30%	<0.01	<0.01

TABLE III: Effect of label smoothing on C/C++ q75 dataset ($\epsilon = 0.1$, $N_t = 10908$)

	without label smoothing			*with label smoothing*			*percentage difference*			*p-value*	
Model Name	METEOR	USE	BLEU	METEOR	USE	BLEU	METEOR	USE	BLEU	METEOR	USE
Attendgru	52.38	62.25	46.50	53.26	62.75	47.41	1.68%	0.80%	1.96%	<0.01	<0.01
Transformer	55.90	65.23	49.30	56.47	65.71	50.15	1.02%	0.74%	1.72%	<0.01	<0.01
AstAttendgru	48.09	57.62	42.14	52.12	61.27	46.43	8.38%	6.33%	10.18%	<0.01	<0.01
Code2seq	16.54	25.67	10.20	17.93	28.31	11.20	8.40%	10.28%	9.80%	<0.01	<0.01
CodeGNNGRU	47.79	57.17	42.13	51.25	60.35	45.63	7.24%	5.56%	8.31%	<0.01	<0.01
AstAttendgru-fc	46.68	56.61	40.79	50.66	60.32	44.62	8.53%	6.55%	9.39%	<0.01	<0.01

TABLE IV: Effect of label smoothing on full Java dataset ($\epsilon = 0.1$, $N_t = 10908$)

	without label smoothing			*with label smoothing*			*percentage difference*			*p-value*	
Model Name	METEOR	USE	BLEU	METEOR	USE	BLEU	METEOR	USE	BLEU	METEOR	USE
Attendgru	35.40	52.84	18.44	35.21	53.08	18.35	-0.54%	0.45%	-0.49%	<0.01	<0.01
Transformer	36.44	53.95	19.12	36.87	54.58	19.36	1.18%	1.17%	1.26%	<0.01	<0.01
AstAttendgru	35.72	53.03	18.53	35.90	53.44	18.77	0.50%	0.77%	1.30%	<0.01	<0.01
Code2seq	35.38	52.96	18.32	35.87	53.45	18.62	1.38%	0.93%	1.64%	<0.01	<0.01
CodeGNNGRU	36.01	53.57	18.91	36.09	53.82	18.89	0.22%	0.47%	-0.11%	0.20	<0.01
AstAttendgru-fc	35.93	53.39	19.14	36.37	53.86	19.61	1.22%	0.88%	2.46%	<0.01	<0.01

TABLE V: Scores of attendgru for different ϵ. Output vocabulary size for the top table is 10k and for the bottom table is 44k.

	metric scores			*t-stat, p-value*	
ϵ	METEOR	USE	BLEU	METEOR	USE
0	32.82	50.21	18.87	-	-
0.001	32.79	50.22	18.85	-0.26, 0.80	0.13, 0.90
0.003	33.01	50.87	18.82	1.34, 0.18	4.60,<0.01
0.007	32.99	50.36	18.99	1.32, 0.19	1.13, 0.26
0.02	32.97	50.36	19.01	1.07, 0.28	0.99, 0.32
0.05	33.30	51.00	19.30	3.18,<0.01	5.19,<0.01
0.10	33.39	51.41	19.45	3.76,<0.01	7.84,<0.01
0.25	33.29	51.05	19.32	2.93,<0.01	5.12,<0.01
0.40	33.34	50.98	19.32	3.25,<0.01	4.63,<0.01

	metric scores			*t-stat, p-value*	
ϵ	METEOR	USE	BLEU	METEOR	USE
0	32.94	50.30	18.94	-	-
0.001	32.94	50.23	18.89	-0.07, 0.94	-0.79, 0.43
0.003	32.79	50.10	18.95	-1.24, 0.21	-1.66, 0.10
0.007	32.71	50.11	18.86	-1.70, 0.09	-1.42, 0.16
0.02	32.85	50.60	18.90	-0.63, 0.53	2.06, 0.04
0.05	32.95	50.86	19.02	0.07, 0.95	3.66,<0.01
0.10	33.20	50.97	19.05	1.83, 0.07	4.37,<0.01
0.25	33.18	50.88	19.05	1.45, 0.15	3.55,<0.01
0.40	33.16	50.89	19.22	1.32, 0.19	3.64,<0.01

V. RQ2 - HYPERPARAMETER TUNING

Methodology: To answer RQ2, we choose four out of the six models used in RQ1 to study in greater detail: `attendgru`, `transformer`, `codegnngru` and `ast-attendgru-fc`. We use these models because they each represent a different family of source code summarizaton models: `attendgru` from simple seq2seq family, `transformer` from self-attention architecture family, `codegnngru` from code + AST represented as GNN in a seq2seq architecture family and `ast-attendgru-fc` from code + flat AST + contextual information in a seq2seq model family. For each model we vary the value of ϵ; we start with 0.001 and increase the value by a factor of e^n for n=1 to 6. Along the way, we also include 0.25 ($0.001 \times e^{5.5}$) for a more granular look as ϵ increases exponentially for large values of n. We run each model for these 8 different configurations on the Java-q90 dataset. We only choose one dataset for RQ2 because we want to eliminate any experimental variables that may be introduced by the dataset, but also due to resource constraints. We choose the Java-q90 dataset because it is part of a peer-vetted and widely used source code summarization dataset. To understand how output vocabulary size affects model performance with label smoothing, we then change the size of output vocabulary more than four-fold (from 10k to 44k) and run each model again. We train each model configuration for eight epochs and choose the model with highest validation accuracy. Similar to RQ1, we evaluate the performance of these models using automated evaluation techniques discussed in Section III-D. Additionally we perform a paired t-test between each configuration with label smoothing and the corresponding prediction without label smoothing to identify the statistical significance for METEOR and USE. Notice again that we do not perform a t-test on BLEU score because BLEU is a corpus level metric.

Key Findings: Our key finding in answering RQ2 is that higher values of ϵ generally lead to higher model performance, although this increase in performance is less pronounced

TABLE VI: Scores of transformer for different ϵ. Output vocabulary size for the top table is 10k and for the bottom table is 44k.

	metric scores			*t-stat, p-value*	
ϵ	METEOR	USE	BLEU	METEOR	USE
0	33.64	51.81	18.99	-	-
0.001	33.53	51.46	19.18	-0.83, 0.41	-2.71, <0.01
0.003	33.27	51.36	19.06	-2.53, 0.01	-3.18, <0.01
0.007	33.77	51.49	19.13	0.81, 0.42	-2.05, 0.04
0.02	33.84	52.57	19.08	1.22, 0.22	4.76, <0.01
0.05	33.65	51.64	19.11	0.07, 0.95	-1.14, 0.26
0.10	33.75	52.25	19.11	0.69, 0.49	2.72, <0.01
0.25	33.72	52.27	18.97	0.50, 0.62	2.96, <0.01
0.40	33.84	52.10	19.19	1.21, 0.23	1.73, 0.08

	metric scores			*t-stat, p-value*	
ϵ	METEOR	USE	BLEU	METEOR	USE
0	33.28	51.55	18.73	-	-
0.001	33.38	50.95	18.85	0.62, 0.54	-3.67, <0.01
0.003	33.09	51.23	18.77	-1.21, 0.23	-2.11, 0.04
0.007	33.30	51.55	18.75	-0.83, 0.41	0.01, 0.99
0.02	33.30	51.82	18.85	0.13, 0.90	1.71, 0.09
0.05	33.32	51.74	18.77	0.27, 0.79	1.21, 0.22
0.10	33.77	52.17	18.94	3.04, <0.01	3.89, <0.01
0.25	33.84	52.51	19.22	3.40, <0.01	5.95, <0.01
0.40	34.23	52.27	19.56	5.57, <0.01	4.33, <0.01

TABLE VII: Scores of codegnngru for different ϵ. Output vocabulary size for the top table is 10k and for the bottom table is 44k.

	metric scores			*t-stat, p-value*	
ϵ	METEOR	USE	BLEU	METEOR	USE
0	32.30	49.37	18.00	-	-
0.001	32.53	49.94	18.22	1.55, 0.12	3.64, <0.01
0.003	32.46	49.76	18.24	0.98, 0.33	2.30, 0.02
0.007	32.18	49.69	17.96	-0.79, 0.43	2.06, 0.04
0.02	32.53	49.88	18.42	1.40, 0.16	3.02, <0.01
0.05	32.65	49.92	18.59	2.16, 0.03	3.22, <0.01
0.10	33.17	50.49	18.76	5.46, <0.01	6.82, <0.01
0.25	33.36	51.08	18.97	6.65, 0.01	10.19, <0.01
0.40	33.37	51.35	19.00	6.51, <0.01	11.79, <0.01

	metric scores			*t-stat, p-value*	
ϵ	METEOR	USE	BLEU	METEOR	USE
0	32.26	49.33	18.20	-	-
0.001	32.56	49.85	18.22	1.87, 0.06	3.31, <0.01
0.003	32.25	49.62	18.19	-0.07, 0.94	1.77, 0.08
0.007	32.75	49.68	18.57	3.02, <0.01	2.10, 0.04
0.02	32.36	49.76	18.34	0.62, 0.53	2.68, <0.01
0.05	32.86	50.31	18.53	3.65, <0.01	5.98, <0.01
0.10	32.71	49.97	18.48	2.64, <0.01	3.77, <0.01
0.25	32.68	49.92	18.53	2.51, 0.01	3.53, <0.01
0.40	32.88	50.50	18.72	3.72, <0.01	7.11, <0.01

for $\epsilon > 0.1$. Furthermore, this trend is unaffected as we increase the vocabulary size from 10k to 44k. Therefore, the value of the smoothed probability per token ($\frac{\epsilon}{N_t - 1}$) does not affect model performance. While one must decide the best hyperparameters for oneself based on prevailing experimental conditions, our recommendation is to set ϵ=0.1 for initial evaluation. Caution is advised for high values of ϵ as it forces models to predict more common words and eliminate rare/unique words. We discuss this issue in greater detail in RQ3.

Table V shows the effect of increasing the label smoothing factor, ϵ for attendgru model. For this architecture, we notice an increase in model performance on all metrics as we increase the value of ϵ from 0.001 to 0.1. METEOR, USE and BLEU scores all achieve the highest score for this configuration of ϵ=0.1. As we increase the output vocabulary size from 10k to 44k, we notice that both METEOR and USE scores decrease slightly for small values of ϵ but then they increase, reaching highest METEOR and USE score for ϵ=0.1.

Table VI shows the effect of increasing ϵ for transformers. We initially see a decrease in metric scores for small values of ϵ. As we increase ϵ, the metric scores start to increase. Interestingly, for the 44k output vocabulary, we see a substantially significant increase in metric scores for all values of $\epsilon >$ 0.05. We attribute this to the fact that higher output vocabulary size increases the amount of rare words in the prediction. Label smoothing helps models generalize by focusing on more commonly occuring words.

Table VII shows the effect of increasing ϵ for codegnngru. For the 10k output vocabulary set, we notice a pattern similar to attendgru and transformers. While USE shows a statistically significant improvement for all cases of ϵ, METEOR only starts demonstrating a statistically significant improvement for all values of $\epsilon > 0.05$. For the 44k output vocabulary set, we see an improvement across the board, that is explained by the model eliminating noise, introduced by the large output vocabulary, set using label smoothing.

Table VIII shows the effect of increasing ϵ for astattendgru-fc. For the 10k output vocabulary set, we see a statistically significant decrease in the model performance for ϵ=0.001. For ϵ=0.003, we see a slight increase in metric scores, although it is not statistically significant. For all but one other configuration, we see a statistically significant increase in metric scores for both METEOR and USE. We see a similar pattern for the 44k output vocabulary set as the metric scores change insignificantly for ϵ=0.001 and ϵ=0.003 but then achieve high improvement for all other configurations. Note that we do not recommend any particular model for the problem of source code summarization. Rather we demonstrate how all existing models benefit in performance from adding label smoothing.

VI. RQ3 - Vocabulary Diversity

Methodology: To answer RQ3, we look at the output vocabulary distribution for all the models in RQ1 and RQ2. We find the total number of words predicted per model as well as the total number of unique words per prediction set. This helps us identify the average word frequency in the prediction set. We further try to identify how label smoothing affects model performance on the action word prediction problem. We re-implement each model discussed in Section III-C to only predict the action word. Related work predict the top-10 or top-40 most commonly occurring action words and bucket the rest under "other." We try to predict all the action words instead to see how the diversity of output vocabulary changes with label smoothing. We train each model for the Java-q90 and C/C++-q75 dataset. We train each baseline three times: once without label smoothing, once with ϵ=0.1 and once with ϵ=0.4. We train each configuration for 10 epochs and

TABLE VIII: Scores of astattendgru-fc for different ϵ. Output vocabulary size for the top table is 10k and for the bottom is 44k.

	metric scores			*t-stat, p-value*	
ϵ	METEOR	USE	BLEU	METEOR	USE
0	33.09	50.05	18.76	-	-
0.001	32.74	49.53	18.66	-2.26, 0.02	-3.24, <0.01
0.003	33.32	50.23	19.07	1.40, 0.16	1.04, 0.30
0.007	33.62	50.60	19.08	3.26, <0.01	3.25, <0.01
0.02	33.49	50.16	19.18	2.40, 0.02	0.64, 0.52
0.05	33.50	50.81	19.41	2.47, 0.01	4.56, <0.01
0.10	33.66	50.91	19.38	3.47, <0.01	5.10, <0.01
0.25	33.80	50.88	19.54	4.21, <0.01	4.76, <0.01
0.40	33.98	51.22	19.58	5.48, <0.01	7.06, <0.01

	metric scores			*t-stat, p-value*	
ϵ	METEOR	USE	BLEU	METEOR	USE
0	32.69	49.87	18.65	-	-
0.001	33.03	49.54	18.96	2.06, 0.04	-1.97, 0.05
0.003	32.89	50.19	18.85	1.24, 0.21	1.93, 0.05
0.007	33.36	50.57	19.07	4.13, <0.01	4.31, <0.01
0.02	33.69	50.73	19.24	6.15, <0.01	5.23, <0.01
0.05	33.58	50.92	19.39	5.42, <0.01	6.37, <0.01
0.10	33.75	51.37	19.35	6.34, <0.01	8.77, <0.01
0.25	34.27	51.95	19.75	9.76, <0.01	12.77, <0.01
0.40	33.84	51.21	19.46	7.21, <0.01	8.18, <0.01

choose the model with the highest validation accuracy score for our comparison (standard practice in related work). We then evaluate the performance of these models using precision, recall and f1-score. This is because action word prediction is a multi-class classification problem, while BLEU, METEOR and USE+c are more suited for full sentence prediction.

Key Findings: Our key finding in answering RQ3 is that label smoothing decreases word diversity in output vocabulary. As we increase ϵ, the total number of words in the output prediction remains similar but the total number of unique words decreases, thus increasing the average output word frequency. We also find that the action word prediction is unaffected by label smoothing. Therefore, we conclude that the improvement in full model performance perceived in RQ1 must be coming from the rest of the sentence.

Figure 2 shows how the number of unique words change with the addition of label smoothing for all the models in RQ1. The blue bar represents models without label smoothing. The orange bar represents the same models with label smoothing (ϵ=0.1). As the figures show, the total number of unique words decrease for all baselines, for all datasets. To further study this trend, we run the same word diversity study on the models in RQ2. Figure 3 shows how the number of unique words change with changing values of the ϵ. A clear pattern emerges regardless of output vocabulary size (solid for 10k output vocabulary size and dashed for 44k output vocabulary size): The total number of unique words decrease steadily as we increase the value of ϵ. It is interesting to note that ast-attendgru-fc experiences highest improvement in performance AND the highest decrease in the number of unique words (29.1% for 10k output and 39.9% for 44k output).

Both figures suggest that label smoothing improves model performance by predicting commonly occurring words more often, thus reducing word diversity. This explains how label smoothing achieves generalization: it avoids predicting rare words from the output distribution to reduce model loss.

To further narrow down how label smoothing improves comment generation, we test its effect on action word prediction problem. Haque *et al.* showed that the action word prediction is a crucial stepping stone problem in source code summarization and models that predict action words poorly also predict the rest of the comment sentence poorly. We introduce label smoothing to the action word prediction problem to see if it improves model performance while reducing the number of unique action words predicted.

Tables IX and X show the effect of label smoothing on action word prediction. We train the models under 3 conditions: ϵ=0 (no label smoothing), ϵ=0.1 and ϵ=0.4. We see that label smoothing has little to no effect on model performance for action word prediction. No model shows an improvement greater than 0.02 between ϵ=0 and ϵ=0.4 for precision, recall or f1-score. Figure 4 shows the distribution of unique action words for each model with each label smoothing factor for the

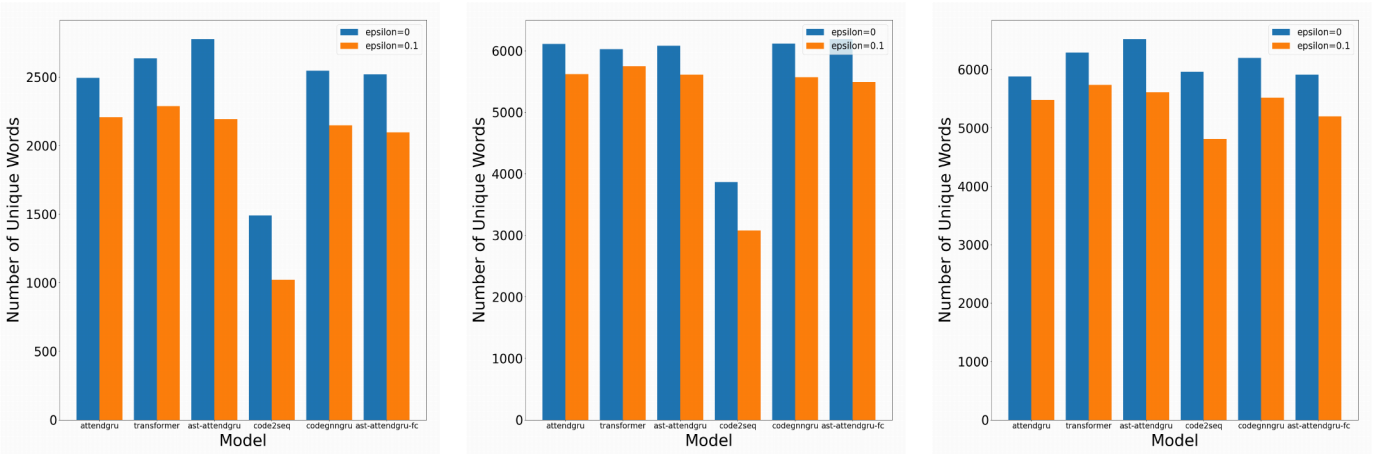


Fig. 2: Comparison of word diversity for baselines with and without label smoothing (ϵ=0.10). Blue indicates baseline total number of unique words. Orange indicates total number of unique words for identical model with label smoothing. The left figure is for the Java-q90 dataset, the center figure is for the C/C++-q75 dataset, and the right figure is for the full Java dataset.

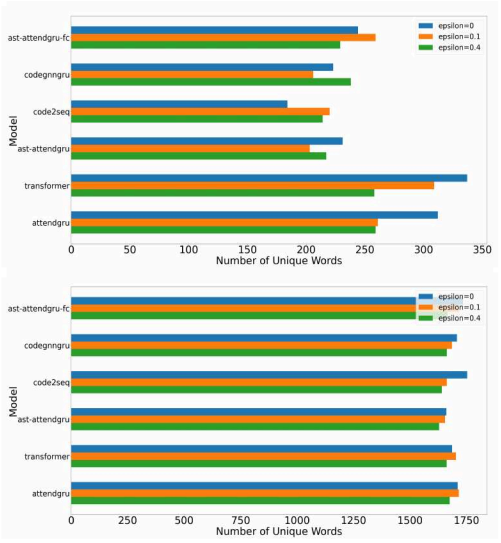
TABLE IX: Results for RQ3 on Java q90 dataset.

Model Name	$\epsilon = 0$			$\epsilon = 0.1$			$\epsilon = 0.4$		
	P	R	F1	P	R	F1	P	R	F1
Attendgru	0.43	0.51	0.45	0.43	0.51	0.45	0.44	0.52	0.46
Transformer	0.45	0.50	0.45	0.45	0.50	0.44	0.46	0.52	0.46
AstAttendgru	0.41	0.49	0.43	0.42	0.50	0.43	0.43	0.50	0.44
Code2Seq	0.41	0.50	0.44	0.41	0.50	0.44	0.42	0.51	0.44
CodeGNNGRU	0.43	0.51	0.45	0.43	0.51	0.44	0.43	0.51	0.45
AstAttendgru-fc	0.44	0.52	0.46	0.45	0.52	0.45	0.45	0.52	0.46

TABLE X: Results for RQ3 on C/C++ q75 dataset.

Model Name	$\epsilon = 0$			$\epsilon = 0.1$			$\epsilon = 0.4$		
	P	R	F1	P	R	F1	P	R	F1
Attendgru	0.69	0.71	0.69	0.70	0.71	0.69	0.71	0.72	0.70
Transformer	0.71	0.72	0.70	0.72	0.73	0.71	0.72	0.73	0.71
AstAttendgru	0.69	0.69	0.67	0.70	0.70	0.68	0.71	0.71	0.68
Code2Seq	0.62	0.63	0.61	0.62	0.62	0.60	0.62	0.62	0.60
CodeGNNGRU	0.69	0.70	0.68	0.70	0.71	0.69	0.70	0.72	0.69
AstAttendgru-fc	0.69	0.70	0.67	0.70	0.71	0.69	0.70	0.72	0.69

Fig. 4: Action word diversity with ($\epsilon = 0.1, 0.4$), without ($\epsilon = 0$) label smoothing.



Java-q90 and C/C++-q75 dataset respectively. We can see that there is no clear trend in the diversity of action words with changing values of ϵ for either dataset: the number of unique action words decrease in some cases and increase in others.

Two key findings can be derived from this: 1) The performance improvement we see in RQ1 and RQ2 is not due to the improved action word prediction, rather most likely coming from predicting the rest of the sentence. 2) Adding label smoothing does not affect the diversity of unique action words. These findings further bolster the intuition that label smoothing improves model generalization by predicting more commonly occurring words in favor of rarer words. This explains why we recommend the use of ϵ=0.1 instead of 0.4 in RQ2. While 0.4 squeezes out the best performance, most models demonstrate statistically significant improvement at 0.1 as well. Further improvement comes at the cost of reduced word diversity.

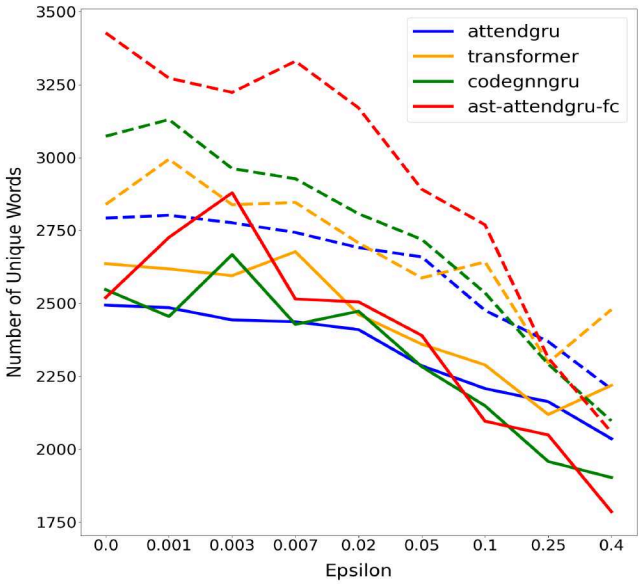


Fig. 3: Comparison of word diversity for baselines with different values of ϵ. The solid lines are trained with 10k output vocabulary and the dashed lines are trained with 44k output vocabulary. Note that the x-axis is logarithmic.

VII. Discussion & Conclusion

This paper contributes to source code summarization research along three major frontiers:

1) We contribute a study in the use of label smoothing as a regularizer in source code summarization. We explore a variety of source code summarization baselines and quantitatively demonstrate that label smoothing improves model performance in terms of three different metrics.
2) We make a recommendation on what label smoothing hyperparameter (ϵ) should be used for source code summarization. We choose four different source code summarization models and tune the value of ϵ for best performance. We identify that higher values of ϵ generally lead to higher model performance. Our recommendation is to use ϵ=0.1 for all source code summarization models. Our justification for this recommendation is in Section V.
3) We contribute new insight in the understanding of how label smoothing generalizes NLG models. We explore the word diversity of output vocabulary for all models trained with and without label smoothing in RQ1 and RQ2. We find that label smoothing reduces the number of unique words predicted in all cases. We further identify that label smoothing does not affect action word prediction. Thus, we posit that for subsequent word prediction in the output sentence, label smoothing improves model performance by predicting commonly occurring words more often and avoiding rare/unique words to reduce model loss.

Appendix: `https://github.com/ls-scs/ls_scs`

Acknowledgment

This work is supported in part by NSF CCF-1452959 and CCF-1717607. Any opinions, findings, and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.

REFERENCES

[1] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 35–44.

[2] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 43–52.

[3] D. Kramer, "Api documentation from source code comments: a case study of javadoc," in *Proceedings of the 17th annual international conference on Computer documentation*. ACM, 1999, pp. 147–153.

[4] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: a survey," in *Proceedings of the 2002 ACM symposium on Document engineering*. ACM, 2002, pp. 26–33.

[5] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2017.

[6] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 795–806.

[7] F. Zhao, J. Zhao, and Y. Bai, "A survey of automatic generation of code comments," in *Proceedings of the 2020 4th International Conference on Management Engineering, Software Engineering and Service Sciences*, 2020, pp. 21–25.

[8] D. Zügner, T. Kirschstein, M. Catasta, J. Leskovec, and S. Günnemann, "Language-agnostic representation learning of source code from structure and context," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=Xh5eMZVONGF

[9] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*. ACM, 2018, pp. 200–210.

[10] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[11] A. LeClair and C. McMillan, "Recommendations for datasets for source code summarization," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019, pp. 3931–3937.

[12] N. B. Toomarian and J. Barhen, "Learning a trajectory using adjoint functions and teacher forcing," *Neural Networks*, vol. 5, pp. 473–484, 1992.

[13] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, pp. 3104–3112, 2014.

[14] S. Haque, A. Bansal, L. Wu, and C. McMillan, "Action word prediction for neural source code summarization," *28th IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2021.

[15] S. Jiang and M. de Rijke, "Why are sequence-to-sequence models so dull? understanding the low-diversity problem of chatbots," in *Proceedings of the 2018 EMNLP Workshop SCAI: The 2nd International Workshop on Search-Oriented Conversational AI*, 2018, pp. 81–86.

[16] R. Müller, S. Kornblith, and G. Hinton, *When Does Label Smoothing Help?* Red Hook, NY, USA: Curran Associates Inc., 2019.

[17] M. Lukasik, S. Bhojanapalli, A. K. Menon, and S. Kumar, "Does label smoothing mitigate label noise?" *arXiv preprint arXiv:2003.02819*, 2020.

[18] L. Yuan, F. E. Tay, G. Li, T. Wang, and J. Feng, "Revisiting knowledge distillation via label smoothing regularization," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 3902–3910.

[19] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 101–110.

[20] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 279–290.

[21] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using ir methods for labeling source code artifacts: Is it worthwhile?" in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 2012, pp. 193–202.

[22] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proceedings of the 36th international conference on Software engineering*. ACM, 2014, pp. 390–401.

[23] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, and G. Canfora, "Automatic generation of release notes," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 484–495.

[24] B. Zhang, E. Hill, and J. Clause, "Towards automatically generating descriptive names for unit tests," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 625–636.

[25] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, 2016, pp. 2073–2083.

[26] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "A neural architecture for generating natural language descriptions from source code changes," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 2017, pp. 287–292.

[27] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 135–146.

[28] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. AAAI Press, 2018, pp. 2269–2275.

[29] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *International Conference on Learning Representations*, 2018.

[30] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *International Conference on Learning Representations*, 2019.

[31] S. Gao, C. Chen, Z. Xing, Y. Ma, W. Song, and S.-W. Lin, "A neural model for method name generation from functional description," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 414–421.

[32] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," *arXiv preprint arXiv:1811.01824*, 2018.

[33] S. Haque, A. LeClair, L. Wu, and C. McMillan, "Improved automatic summarization of subroutines via attention to file context," *International Conference on Mining Software Repositories*, 2020.

[34] R. Haldar, L. Wu, J. Xiong, and J. Hockenmaier, "A multi-perspective architecture for semantic code search," *arXiv preprint arXiv:2005.06980*, 2020.

[35] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *28th ACM/IEEE International Conference on Program Comprehension (ICPC'20)*, 2020.

[36] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *arXiv preprint arXiv:2005.00653*, 2020.

[37] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid {gnn}," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=zv-typ1gPxA

[38] A. LeClair, A. Bansal, and C. McMillan, "Ensemble models for neural source code summarization of subroutines," *arXiv preprint arXiv:2107.11423*, 2021.

[39] S. Gao, C. Gao, Y. He, J. Zeng, L. Y. Nie, and X. Xia, "Code structure guided transformer for source code summarization," *arXiv preprint arXiv:2104.09340*, 2021.

[40] Y. Wang, E. Shi, L. Du, X. Yang, Y. Hu, S. Han, H. Zhang, and D. Zhang, "Cocosum: Contextual code summarization with multi-relational graph neural network," *arXiv preprint arXiv:2107.01933*, 2021.

[41] A. Bansal, S. Haque, and C. McMillan, "Project-level encoding for neural source code summarization of subroutines," *arXiv preprint arXiv:2103.11599*, 2021.

[42] Z. Gong, C. Gao, Y. Wang, W. Gu, Y. Peng, and Z. Xu, "Source code summarization with structural relative position guided transformer," *arXiv preprint arXiv:2202.06521*, 2022.

[43] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[44] I. Sutskever, J. Martens, and G. E. Hinton, "Generating text with recurrent neural networks," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 1017–1024.

[45] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[46] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," 2014.

[47] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[48] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2818–2826.

[49] J. Lienen and E. Hüllermeier, "From label smoothing to label relaxation," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 10, pp. 8583–8591, May 2021. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/17041

[50] C.-B. Zhang, P.-T. Jiang, Q. Hou, Y. Wei, Q. Han, Z. Li, and M.-M. Cheng, "Delving deep into label smoothing," *IEEE Transactions on Image Processing*, vol. 30, pp. 5984–5996, 2021. [Online]. Available: https://doi.org/10.1109%2Ftip.2021.3089942

[51] L. Xie, J. Wang, Z. Wei, M. Wang, and Q. Tian, "Disturblabel: Regularizing cnn on the loss layer," 2016. [Online]. Available: https://arxiv.org/abs/1605.00055

[52] S. Stapleton, Y. Gambhir, A. LeClair, Z. Eberhart, W. Weimer, K. Leach, and Y. Huang, "A human study of comprehension and code summarization," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 2–13.

[53] R. Xie, W. Ye, J. Sun, and S. Zhang, "Exploiting method names to improve code summarization: A deliberation multi-task learning approach," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, 2021, pp. 138–148.

[54] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019, pp. 143–153.

[55] Z. Eberhart, A. LeClair, and C. McMillan, "Automatically extracting subroutine summary descriptions from unstructured comments," *arXiv preprint arXiv:1912.10198*, 2019.

[56] D. M. Powers, "Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation," 2011.

[57] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 2002, pp. 311–318.

[58] D. Roy, S. Fakhoury, and V. Arnaoudova, "Reassessing automatic evaluation metrics for code summarization tasks," in *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[59] S. Haque, Z. Eberhart, A. Bansal, and C. McMillan, "Semantic similarity metrics for evaluating source code summarization," 2022. [Online]. Available: https://arxiv.org/abs/2204.01632

[60] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005, pp. 65–72.

[61] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," *Text Summarization Branches Out*, 2004.

[62] D. Cer, Y. Yang, S. yi Kong, N. Hua, N. Limtiaco, R. S. John, N. Constant, M. Guajardo-Cespedes, S. Yuan, C. Tar, Y.-H. Sung, B. Strope, and R. Kurzweil, "Universal sentence encoder," 2018.