CodeSumBart: Metric-Oriented Pretraining for Neural Source Code Summarisation Transformers

Anonymous submission

Abstract

Source code summaries give developers and maintainers vital information about source code methods, however writing these summaries takes up the developers' time and these summaries are often missing, incomplete, or outdated. Neural source code summarisation solves these issues by summarising source code automatically. Current solutions use Transformer neural networks to achieve this. We present CodeSumBart - a $\text{Bart}_{\text{Base}}$ model for neural source code summarisation, pretrained on a dataset of Java source code methods and English method summaries. We present a new approach to training Transformers for neural source code summarisation. Our training makes use of an NLG metric to validate each epoch of training, then uses validation results to change model weights during the training process to produce a better model. Our results show that while English and Java share many words, the syntax and grammar of the two languages differ to the extent that pretraining a model on English language data does not aid models in understanding Java, and may even have a detrimental effect on Java understanding. We found that in our approach, using larger n-gram precision Bleu metrics for epoch validation, such as Bleu-4, produces better performing models than other common NLG metrics.

Keywords: summarisation, source code, optimisation

1. Introduction

Neural Source Code Summarisation (NSCS) aims to summarise source code methods without developer interaction, using neural network models. NSCS models require extensive training on large datasets of source code and related summaries to produce outputs with often low similarity to human-written summaries. Our training produces a model which produces better outputs while requiring no more training than other, similar -sized models. NSCS has grown in recent years with the development of new task-specific models, many of which build on the Transformer architecture (Vaswani et al., 2017). A key example is Ahmad et al., 2020's NeuralCodeSum model, which uses a Transformer, trained on the TL-CodeSum dataset (Hu et al., 2018) for a maximum of 200 epochs, with an early stop if there is no improvement after 20 epochs. Another is CodeBert (Feng et al., 2020), which is a Bert language model built using Transformer encoders.

When training Transformer models for summarisation tasks, each epoch of training can be validated against a Natural Language Generation (NLG) metric. NLG metrics are often calculated alongside a loss metric or loss function, which is used to optimise the model. Our training method takes a different approach by removing the reliance on a loss function. Instead, we rely on the NLG metric to enhance the model's training for a given task, in order to simplify the training process while retaining a better model, as shown in Table 6. Our training process is detailed in Section 3. Validation with loss or NLG metrics allows for "checkpointing" where the improvement in outputs from each epoch of training can be compared to previous epochs and the training can be stopped early if the training is no longer improving. The use of early stopping and checkpointing prevents overfitting to a given dataset by ensuring the outputs remain generic.

Bleu (Papineni et al., 2002) evaluates the quality of a predicted text, based on matching n-grams from a reference text, through the use of a computed precision and brevity penalty. Bleu-1 refers to Bleu with unigram precision, Bleu-2 with bigram precision, and so on. Lin and Och (2004) proposed a method of smoothing BLEU for use at a sentence level, to ensure that for Bleu-*n*, sentences with fewer than *n* tokens still receive a positive score. Similar common metrics for evaluating the quality of a summary include Rouge (Lin, 2004) and Meteor (Denkowski and Lavie, 2014). While Smoothed Bleu-4 is the most common metric for validating training epochs for NSCS Transformers, any of these metrics could be selected.

Recent work in Large Language Model (LLM)-based metrics also present new possibilities for validating the training of models for source code summarisation. These metrics aim to improve the reliability of scores by using LLMs to capture the semantics of a given text through the use of contextual embeddings. Zhang et al. (2019) introduced LLM-based metrics in 2019, with BertScore, which remains the basis for other new LLM-based metrics, such as FrugalScore (Kamal Eddine et al., 2022). LLM-based metrics come with a requirement for far more significant computational resources compared to n-gram-based metrics, and Kamal Eddine et al. (2022) attempt to solve this with FrugalScore by reducing the number of parameters required while retaining accuracy.

We train a Bart Transformer model (Lewis et al., 2020) on a source code summarisation task using a variety of validation metrics. We present a method of optimising pretraining to provide better results by monitoring the validation metric used, and checkpointing the best performing epoch. When an epoch fails to improve, the model weights are reverted to the best performing epoch, and the training continues. After 5 training epochs have failed to improve and a minimum of 20 training epochs have taken place, training stops. We discuss this in detail in Section 3.

1.1. Research Questions

RQ.1 Does pretraining on English language data improve model effectiveness for source code summarisation?

To answer this question, we fine-tune two pretrained transformer models commonly used for English summarisation tasks on our source code summarisation task. We then evaluate these against a suite of NLG metrics. Following this, we pretrain the same two models with randomly initialised weights on our source code summarisation task.

RQ.2 Does validating a model on LLM-based metrics improve the model's predictions over validating it on traditional, n-gram-based NLG metrics?

To answer this question, we compare the overall metric results of those models validated using n-gram-based metrics to those using BERTScore (Zhang et al., 2019) and FrugalScore (Kamal Eddine et al., 2022) to see if there is an improvement in model training provided by using LLM-based metrics. A measurable improvement caused by using LLM-based metrics for validation, rather than n-gram-based metrics shows that LLM-based metrics' improved ability to capture semantics allow them to aid in generating better models for automatic source code summarisation.

RQ.3 Does validating on a common NLG metric from Table 2 cause the model to perform better on NSCS?

We report whether any one metric is better for validation (producing a model that gives more accurate outputs) than others. Models such as NeuralCodeSum (Ahmad et al., 2020) use Smoothed BLEU-4 by default, but there is a wide variety of available metrics which can be used. A measurable improvement in the quality of outputs when the model is evaluated against a series of metrics means that this technique has the potential to be used in generating better models for automatic source code summarisation.

1.2. Contributions

We propose a new approach to the training and validation of Transformer models for NSCS tasks, which improves the quality of outputs, when compared to similar models, without a significant increase in the size or training time of a model. We present CodeSumBART, a $\text{BART}_{\text{BASE}}$ model, utilising this training approach to automatically summarise Java source code.

2. Dataset

In order to train, validate, and evaluate the models, we use the filtered version of LeClair and McMillan, 2019's Funcom dataset of Java source code method - English language summary pairs, as done in previous works by Mahmud et al. (2021) and Phillips et al. (2022). We clean the dataset following Phillips et al., 2022's approach, using their official Java implementation of the dataset cleaning tool[1].

Phillips et al., 2022's method cleans the dataset using the matched pairs of Java source code and JavaDoc comments. The cleaning method uses JavaParser (van Bruggen et al., 2020) to select only compilable Java code and remove inline code comments. It then finds the method summaries from the JavaDoc by extracting the first line of text with more than eight characters. We then follow Phillips et al., 2022's steps: remove HTML and special characters (characters which are not alphanumeric, full-stops, apostrophes, or white space) from the summary and lowercase it. Repeated method-summary pairs are then removed from the dataset, which is trimmed from 1.2 million pairs to roughly $500{,}000$ pairs and split randomly into 80% training, 10% validation, and 10% evaluation datasets. This is the same split used by Ahmad et al. (2020), Mahmud et al. (2021), and Phillips et al. (2022).

Table 1: Split of methods in the dataset.

Training	Validation	Evaluation
$399{,}999$	$49{,}999$	$49{,}999$
80%	10%	10%

Our final dataset contains $499{,}997$ method-summary pairs from multiple projects, split randomly into training, validation, and evaluation, as per Table 1.

3. Research Methodology

We began by selecting the metrics we would use for validating models during training and evaluating models. The metrics chosen are as shown in Table 2: We selected BLEU-1-4, as well as Smoothed BLEU-4. BLEU-1 is a metric frequently used for

[1] Phillips et al., 2022's dataset cleaning tool is found at github.com/phillijm/JavaDatasetCleaner

evaluating summarisation, and Smoothed BLEU-4 is the metric employed for epoch validation by previous work by Ahmad et al. (2020) and Feng et al. (2020). We chose to add BLEU-2-4, as did Phillips et al. (2022), in order to observe the effect of changing the n-gram precision of the metric, as well as observe any effect smoothing had. METEOR can also be used used to evaluate source code summarisation, and is reported by Ahmad et al. (2020), Mahmud et al. (2021), and Phillips et al. (2022). All three of these papers also use ROUGE-L for the same purpose. We propose using ROUGE-1 also, in order to observe whether validating a model on ROUGE using the co-occurrence of unigrams in a text provides better or worse outputs than ROUGE-L, which analyses the longest common subsequence of tokens in a text.

In addition to these common summarisation metrics, we measure FrugalScore and BERTScore, which utilise LLMs to compare if the meaning of a machine-generated text matches the meaning of a human-written one, rather than whether the language used matches. LLM-based metrics achieve this by capturing contextual embeddings. During

Table 2: Metrics used.

Metric
BLEU-1-4 & SMOOTHED BLEU-4
METEOR
ROUGE-1 & -L
FrugalScore
BERTScore

our model training, we validate each epoch on a given NLG metric from Table 2. We use this metric to better optimise the performance of our model to the task by checkpointing the best epoch and reverting epochs that did not show improvement. When an epoch shows improvement in the metric, it is checkpointed as the best model; when an epoch fails to show improvement in the metric, the model weights are reverted to the weights of the best performing epoch from these checkpoints before continuing training. Our use of NLG metrics is similar to how a loss function can be used in training to optimise a given model. We also use checkpoints for early stopping the model training. When a minimum threshold of 20 training epochs have taken place, if five consecutive epochs fail to provide any improvement to the model, we stop training in order to prevent overfitting. In this experiment, we also implemented a maximum of 200 training epochs for the same purpose, but did not reach this limit in any of our training.

Our training and validation process is shown in Figure 1. Our training dataset split of $399{,}999$ method-summary pairs is used in the training step. As we validate our model, we use a validation split of $49{,}999$ pairs. We use this data to calculate an NLG metric, then compare the average metric result to previous validation steps. If the model has improved in the last 5 epochs (early-stopping mechanism, x in Figure 1) and the model produced the highest average metric score this epoch, these model weights are saved as a checkpoint, and the next epoch of training begins unless the maximum number of training epochs (n in Figure 1) has been reached. If the model has shown improvement in the past 5 epochs, but has not improved in this training epoch, the model weights are reverted to the best scoring checkpoint. When this takes place, a small amount of noise is added to the weights, in order to better prevent overfitting to the dataset and to prevent the model from generating the same unsuccessful model weights as the previous attempt. If the model has not improved in the last 5 epochs, the early stopping mechanism is called. When the early stopping mechanism is called, or the maximum number of training epochs has been reached, we evaluate the model against all of our NLG metrics, using the evaluation dataset split of $49{,}999$ method-summary pairs. In order to ensure reliable results, we also required a minimum of 20 training epochs. The results of our model evaluation can be found in Tables 4, 5, and 6.

3.1. Methodology for RQ.1

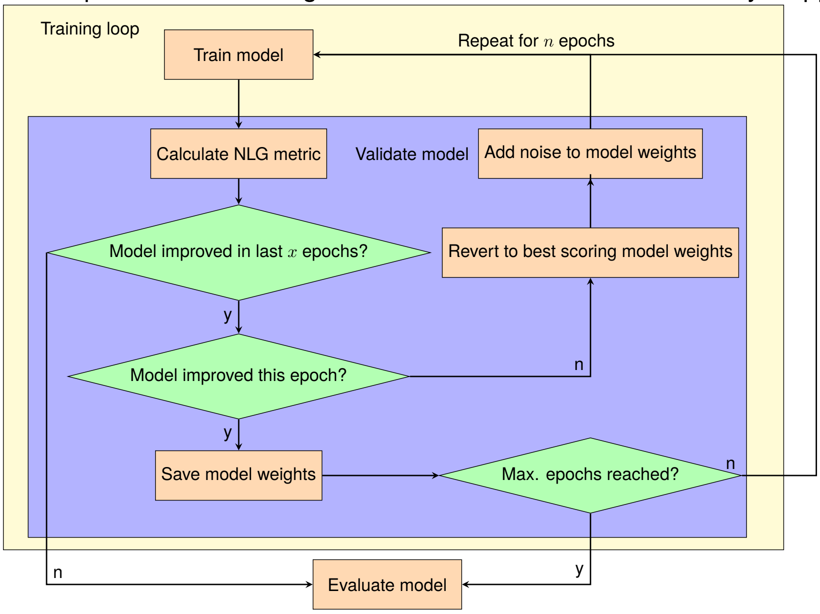
We selected two transformer models commonly used for summarisation tasks: T5 (Raffel et al., 2020) and BART (Lewis et al., 2020). We selected these models due to their popularity, with each model having a high number of citations on Google Scholar and a high number of downloads on HuggingFace, and the availability of low resource usage versions of the model, T5_{SMALL} and $\text{BART}_{\text{BASE}}$, allowing us to train models on machines which are commercially available with a low environmental impact.

The T5_{SMALL} pretrained model is trained on the Colossal Clean Crawled Corpus (C4), proposed in the same paper as the T5 model (Raffel et al., 2020). C4 is a large English dataset, containing roughly 800GB of data extracted from the Common Crawl[2] archive of text mined by crawling the web. The $\text{BART}_{\text{BASE}}$ pretrained model is trained on a variety of tasks across several popular English datasets, including the CNN/Daily Mail and XSum summarisation datasets.

We fine-tuned these two pretrained models on our source code summarisation task as described in Section 3 and shown in Figure 1. We also trained models of the same model architecture, without English language pretraining and with randomly initialised weights, on the same task. We

[2]commoncrawl.org

Figure 1: Epoch-based training with NLG metric orientation and early stopping



trained the models on a machine using an `Intel Xeon E5-2650 v4 CPU`, `94GB RAM`, and 4 `NVIDIA Tesla P100 GPUs` running Python 3.9.16 with the Open Cognitive Environment on Ubuntu 22.04.2 LTS. For RQ.1, we used BLEU-1 as our validation metric, due to its simplicity. We then compare these four models to ascertain whether either model architecture is better than the other for source code summarisation, and to observe the effect of English language pretraining on the model's ability to summarise source code.

3.2. Methodology for RQ.2

We selected the best performing model from the model training described in Section 3.1 ($\text{BART}_{\text{BASE}}$, with randomly initialised weights). Following the training method described previously, we trained a series of $\text{BART}_{\text{BASE}}$ models, each one validated on a different metric from Table 2. Once the models were trained, we evaluated each of them against the evaluation dataset split on our full list of NLG metrics in order to establish what effect, if any, the validation metric has had on our model.

In order to establish a baseline to compare our validation and training method against, we also trained the same $\text{BART}_{\text{BASE}}$ model on our dataset, but without any metric used for validation. In this baseline model, loss is calculated during the validation stage and used for checkpointing and early-stopping of the training, but model weights are not changed based on the outcome of this loss. We again used a maximum of 200 training epochs, and a minimum of 20, with early stopping after 5 unsuccessful training epochs. The difference between this baseline training method and our own is the lack of adjusting model weights after validation to match those of the most successful training epoch.

We then compared the results of evaluating all of our models, highlighting the best results from our findings in Table 5. We sought to identify any patterns in the effect that the choice of validation metric had on our training method, as well as to identify whether using Large Language Model (LLM)-based NLG metrics in our approach is able to outperform traditional N-gram-based metrics.

3.3. Methodology for RQ.3

Following on from our findings in Section 4.2 relating to RQ.2, we identified any validation metric which has caused the model to outperform the models validated on other metrics by evaluating each model on the evaluation dataset split, using all metrics listed in Table 2. We present our best-performing model and compare it to other models in use for NSCS tasks, in order to show the improvement our model presents compared to other solutions, in Table 6.

Table 3: Split of methods in the CodeSearchNet dataset.

Training	Validation	Evaluation
164,775	5175	10,948
91%	3%	6%

To test for overfitting to our dataset, We then compared our model to other models on a different dataset, CodeSearchNet (Husain et al., 2019). We cleaned the CodeSearchNet dataset, following the method Phillips et al. (2022) used for Funcom (LeClair and McMillan, 2019). We trimmed the dataset to valid Java methods only, then removed repeat entries. We then stripped HTML data from source code comments and extracted the method summaries from them. We then lowercased and removed special characters from the summaries and stripped out newline characters ("\n") from both methods and summaries. As the dataset is pre-split into testing, validation, and evaluation splits, we maintained these splits. The size of dataset splits for CodeSearchNet can be found in Table 3. We used the evaluation split of $10{,}948$ method-summary pairs in our evaluation of the various models.

The source code used to train each of our models can be found on GitHub[3]. Each model took between 2 - 4 days to run on one GPU, with the exception of the model trained using METEOR, which took approximately a week, being constrained by file read/write speeds due to the nature of the script used to interface with the METEOR metric.

Once we had completed this evaluation, and compared our model to others within the domain of Neural Source Code Summarisation, we trained our model on the WMT 2016 DE-EN machine translation task (Bojar et al., 2016), and evaluated it against the same selection of metrics to gain insight into the generalisability of these methods when training models for tasks other than NSCS. For this task, we used the original split of data of $4{,}548{,}884$ training pairs, 2168 validation pairs, and 2998 evaluation pairs as provided by the dataset, with results shown in Table 7.

4. Result Analysis

4.1. Results Relating to RQ.1

As shown in Table 4, $\text{BART}_{\text{BASE}}$ consistently outperforms T5_{SMALL} for our source code summarisation task. In answer to RQ.1: for BART, the model with randomly initialised weights outperformed the one with pretraining on a corpus of English data when trained and evaluated on our source code summarisation task. For T5, the improvement caused by pretraining was insignificant in comparison to the difference between the two models.

We suspect this is due to a mixture of three factors. First: the nature of the language used to summarise source code, as technical and detailed language, which differs from much of the language used in pretraining, being news and conversational language. Also, the source code summarisation task requires the model to produce English outputs from a Java input text, whereas pretraining tasks on English language corpora require the model to produce English outputs from English inputs. Our results show that while English and Java share many words, the syntax and grammar of the language differ enough that pretraining models on English data does not aid models in understanding Java. Finally, the architecture of the models themselves: T5_{SMALL} makes use of 60 million parameters, whereas $\text{BART}_{\text{BASE}}$ uses 140 million.

4.2. Results Relating to RQ.2

After training and validation were complete, we evaluated each of the models on our evaluation dataset split against the ten metrics. We found, from our evaluation results in Table 5, that training the model using BLEU-2-4 metrics, including Smoothed BLEU-4 provides the best-performing models on our dataset, with negligible difference between the models based on the choice of BLEU. The model trained using BLEU-1 in validation performs less well than non-unigram BLEU metrics. Models trained using METEOR and ROUGE-1 perform similarly, with ROUGE-L marginally outperforming BLEU-1.

Our results show that training models using BERTScore or FrugalScore as a validation metric in our training outperforms training without validation and optimisation, but does not perform as well as training using traditional non-unigram n-gram-based metrics for validation. Further work is yet to be done to ascertain why this appears to be the case. We suspect that due to these metrics reliance on embeddings, rather than matching n-grams, key words and phrases may be neglected in generating summaries, leading to less accurate summaries being generated.

4.3. Results Relating to RQ.3

We note, from Table 5, that validation using the BLEU-4 metric provides the best results, while Smoothed BLEU-4 performs similarly to BLEU-2. From our testing, use of larger n-gram BLEU metrics in validation appears to produce more accurate results, however, further work is needed to show whether there is a cut-off at which this is no-longer the case.

We found that the model trained with validation using ROUGE-1 is similar to ROUGE-L; both models achieving similar evaluation results across all metrics. Both ROUGE metrics, when used in validation, produced models slightly less accurate than BLEU-1 when evaluated. In our evaluation, the model trained using METEOR in validation outperformed models trained using ROUGE, but was similarly outperformed by BLEU-1.

We then evaluated our best model against $\text{BART}_{\text{BASE}}$ and two NeuralCodeSum models; one pretrained following Ahmad et al., 2020's methodology, and one pretrained following Phillips et al.,

[3]GitHub: anonymous.4open.science/r/CodeSumBART-0F0E

Table 4: Effects of English Pretraining

Model	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Sm. BLEU-4	METEOR	ROUGE-1	ROUGE-L	FrugalScore	BERTScore
Pretrained T5$_{SMALL}$	50.23	37.74	29.88	24.69	24.98	21.76	53.15	52.01	71.77	68.75
T5$_{SMALL}$*	49.39	36.67	28.71	23.48	23.78	21.18	52.15	50.95	71.21	67.97
Pretrained BART$_{BASE}$	51.87	39.45	31.53	26.22	26.50	23.28	53.89	52.70	72.50	70.23
BART$_{BASE}$*	52.74	40.46	32.62	27.33	27.59	23.84	54.87	53.67	73.12	70.75

* Models with weights randomly initialised

Table 5: Comparison of Evaluation Metrics

Metric	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Sm. BLEU-4	METEOR	ROUGE-1	ROUGE-L	FrugalScore	BERTScore
None (Baseline)*	41.77	26.65	17.89	12.71	13.15	16.62	42.92	41.20	64.25	62.09
BLEU-1	52.74	40.46	32.62	27.33	27.59	23.84	54.87	53.67	73.12	70.75
BLEU-2	53.68	42.30	35.05	30.13	30.38	24.88	55.58	54.56	73.61	71.45
BLEU-3	54.06	42.96	35.88	31.06	31.30	25.25	56.04	55.04	73.86	71.70
BLEU-4	53.58	42.40	35.27	30.41	30.66	24.96	55.56	54.51	73.59	71.70
Smoothed BLEU-4	54.24	43.09	36.03	31.23	31.47	25.27	56.06	55.01	73.48	71.20
METEOR	53.29	41.74	34.37	29.35	29.61	24.59	55.37	54.29	73.42	71.15
ROUGE-1	53.10	41.26	33.69	28.56	28.82	24.24	55.19	54.09	73.32	71.28
ROUGE-L	52.98	40.87	33.12	27.91	28.17	24.06	55.07	53.91	73.28	70.96
FrugalScore	47.63	33.85	25.49	20.13	20.45	20.27	49.52	47.92	69.86	67.53
BERTScore	52.80	40.57	32.77	27.49	27.76	23.90	54.84	53.66	73.14	71.14

* loss is calculated during validation and used for early stopping, but model weights are not reverted.

2022's methodology, as well as CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2021). We evaluated it against two NSCS tasks: our task, derived from the Funcom Dataset (LeClair and McMillan, 2019), and the evaluation task from Husain et al., 2019's CodeSearchNet dataset.

On our task, our model significantly outperformed both NeuralCodeSum models as well as both CodeBERT and GraphCodeBERT across all evaluation metrics, all of which outperform BART$_{BASE}$.

We then processed the Evaluation split of the Java dataset from Husain et al., 2019's CodeSearchNet task. We processed this using Phillips et al., 2022's dataset cleaning tool. Evaluating these models against the CodeSearchNet task, we found our model outperforms the NeuralCodeSum models and BART$_{BASE}$ across all metrics, and outperforms all other models tested in 4 out of 10 metrics, with CodeBERT scoring highest on 5 out of 10 and GraphCodeBERT outperforming other models when evaluated against ROUGE-1 . These results can be seen in Table 6.

Our model-generated outputs have a high mean Word Error Rate (WER) (Popović and Ney, 2007) of approximately 56.6, despite a high BLEU-4. A high WER, (in turn, derived from Levenshtein distance) (Levenshtein et al., 1966), shows that while BLEU shows our model has generated key 4-gram phrases which match the human-written summaries of a method, the structuring of the sentence is unique. Previous work by El-Haj et al. (2014) used WER as a metric to compare pairs of texts as a measure of similarity between two texts. We use WER to compare prediction and reference texts for source code summaries. Example outputs and WERs can be seen in Appendix A.

When we trained our model on the WMT 2016 DE-EN translation task (Bojar et al., 2016), we found that our model provided results (seen in Table 7) which are similar to our model when trained and evaluated on our NSCS task. These results suggest that our methods can successfully be applied to model training in other domains, outside of NSCS.

4.4. Statistical Correlation of Results

Using the evaluation metrics from Table 2, we evaluated each output our model produced on the evaluation split from our dataset. We then used Spear-

Table 6: Comparison of Source Code Summarisation Models Using two Datasets

Evaluated against Funcom (LeClair and McMillan, 2019)

Model	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Sm. BLEU-4	METEOR	ROUGE-1	ROUGE-L	FrugalScore	BERTScore
CodeSumBART	**53.58**	**42.40**	**35.27**	**30.41**	**30.66**	**24.96**	**55.56**	**54.51**	**73.59**	**71.70**
$\text{BART}_{\text{BASE}}$	3.16	1.20	0.39	0.07	0.28	4.83	9.06	7.82	43.80	31.60
NeuralCodeSum	24.07	10.46	4.99	2.67	2.67	8.75	24.45	22.82	53.28	59.95
NeuralCodeSum*	33.71	26.82	22.77	20.30	20.30	19.11	45.68	44.36	64.66	69.02
CodeBERT	23.06	10.49	4.77	1.93	19.33	15.72	38.42	36.71	60.86	67.30
GraphCodeBERT	24.04	10.38	4.62	1.89	19.35	13.84	38.36	36.70	60.75	66.78

Evaluated against CodeSearchNet (Husain et al., 2019)

Model	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Sm. BLEU-4	METEOR	ROUGE-1	ROUGE-L	FrugalScore	BERTScore
CodeSumBART	**27.52**	**14.57**	**8.47**	**5.02**	5.71	10.85	32.05	30.14	60.20	56.97
$\text{BART}_{\text{BASE}}$	3.08	1.12	0.37	0.09	0.23	5.14	9.23	7.41	47.65	30.18
NeuralCodeSum	19.96	7.95	3.73	2.02	2.02	7.64	21.24	19.04	52.83	58.98
NeuralCodeSum*	2.49	1.50	0.98	0.71	0.71	5.71	20.11	19.13	50.73	52.79
CodeBERT	24.30	13.13	6.94	3.94	**17.96**	**12.55**	38.34	**36.13**	**62.23**	**68.37**
GraphCodeBERT	38.42	12.69	6.20	3.22	17.50	12.31	**38.42**	36.11	62.19	68.15

* A NeuralCodeSum model pretrained following Phillips et al., 2022's methodology.

Table 7: CodeSumBART trained on WMT 2016 DE-EN dataset

Metric	Result
BLEU-1	66.67
BLEU-2	54.13
BLEU-3	44.38
BLEU-4	36.57
Smoothed BLEU-4	36.66
METEOR	35.82
ROUGE-1	67.09
ROUGE-L	66.01
FrugalScore	83.37
BERTScore	80.10

man's Rank Correlation Coefficient, ρ, to check the correlation between each metric. We found a strong, positive correlation between all metrics even when the sample size is reduced to a 1% random sample of the data. The lowest value of Spearman's rank correlation coefficient was 0.71 between BERTScore and BLEU-4, this pair also provided our largest p-value: $8.87 * 10^{-71}$ - suggesting a statistically significant result. These results can be seen in Appendix B. The high correlation shows agreement between the metrics; predictions rated highly by one metric are rated highly by the others, suggesting that these metrics are approximately equally capable of evaluating NSCS tasks.

5. Related Work

The Transformer neural network model was introduced by Vaswani et al. (2017) as a general-purpose neural network, initially tested on the WMT 2014 English to German translation task. Since then, the Transformer has become a ubiquitous model for many NLP tasks. Much work has been done to advance the Transformer model; BERT (Devlin et al., 2019) uses bidirectional Transformer encoders in order to produce a language model that can be fine-tuned simply. BART (Lewis et al., 2020) uses an architecture which combines both bidirectional and auto-regressive transformers to build a model. Raffel et al. (2020) introduced T5, a simple transformer model, which treats all tasks as text-to-text problems, using transfer learning to achieve this.

Optimising model training by optimising a model's parameters with respect to evaluation metrics is a concept initially developed by Shen et al. (2016) in the form of Minimum Risk Training (MRT). Shen et al. (2016) apply MRT to machine translation. MRT aims to optimise model parameters by minimising loss in terms of evaluation metrics. Ayana et al. (2016) expand on this principle, and apply minimum risk training to a summarisation task, namely Headline Generation. Norouzi et al. (2016) present an alternative algorithm, Reward Augmented Maximum Likelihood (RML). RML builds on more common maximum likelihood estimation, but adds in a step whereby log-likelihood is optimised based on rewards for possible outputs.

Recent works have applied the Transformer model architecture to NSCS. CodeBERT (Feng et al., 2020) and NeuralCodeSum (Ahmad et al., 2020) use Transformer-based models to summarise source code, with CodeBERT being a bidirectional Transformer model, based on BERT and RoBERTa (Liu et al., 2019). Mahmud et al. (2021) compare these two Transformer models, as well as Code2Seq (Alon et al., 2018) on the Funcom dataset (LeClair and McMillan, 2019). Phillips et al. (2022) establishes a method of Funcom to allow for better training and evaluation of a NeuralCodeSum model, as well as introducing the use of an LLM-based metric for evaluating NSCS. Recent work by Haque et al. (2023) focuses on altering the training process to produce better models for NSCS tasks by using label smoothing. Zhou et al. (2023) propose an alternative improved training approach for models for NSCS tasks by using "meta-learning" to transform the training process into a few-shot deep learning task.

Shi et al. (2022) compare different methods for evaluating NSCS, including a comparison of common implementations of BLEU, and a comparison of different pre-processing operations used for training and evaluating NSCS models. In contrast, Stapleton et al. (2020) take a human approach to evaluating source code summarisation. Stapleton et al. (2020) found that "data suggests that participants did not see a clear difference in quality between human-written and machine generated comments". This paper finds developers' ratings to be an unreliable predictor of how much a summary helps them - and that developer intuition can align poorly with reality when assessing the relevancy of information, while also finding BLEU, and ROUGE to only be weakly correlated with rater-assessed correctness.

Traditional NLG metrics for validating and evaluating the language a model produces include BLEU (Papineni et al., 2002) and Smoothed BLEU (Lin and Och, 2004). Other common NLG metrics, more specialised to summarisation tasks include Lin, 2004's ROUGE and Banerjee and Lavie, 2005's METEOR. METEOR's latest iteration (Denkowski and Lavie, 2014) being the version used in our research.

Large Language Models have increasingly been used to generate metrics for NLG tasks. BERTScore (Zhang et al., 2019) and MoverScore (Zhao et al., 2019) being two examples of these metrics. LLM-based metrics capture contextual embeddings in order to capture and compare the meaning of reference and prediction texts, rather than comparing the n-grams within them. These are large models, with a sizeable environmental impact when implemented at large scale. Kamal Eddine et al., 2022's FrugalScore seeks to solve this by reducing the number of parameters used while retaining accuracy. FrugalScore learns from the internal mapping of LLMs to produce a far smaller language model, capable of similar accuracy. The FrugalScore model we have used is the default model ($\text{BERT}_{\text{TINY}}$ learning from $\text{BERT}_{\text{BASE}}$ on BERTScore).

6. Conclusion

We present CodeSumBART, an improved Transformer model for automatic source code summarisation. Our model uses a new training method to achieve a high degree of accuracy by validating the results of each training epoch against an NLG metric and using the performance against that metric to revert the model weights from poorly-performing training epochs to those from the best-performing training epoch.

We have shown how NLG metrics can be used in validation to generate a better-performing model during the training process. Our findings show that our training provides an improved method of training transformer models for automatic source code summarisation. CodeSumBART outperforms state-of-the-art models in evaluation across many metrics and produces outputs comparable to human-written summaries to within a high degree of accuracy in two Java source code summarisation tasks.

7. Limitations

In this paper, we have only used a dataset for the summarisation of Java source code in English. Further research is required to establish the validity of our results in the setting of other languages, particularly our findings for RQ.1, with respect to whether transformer models pretrained on English data perform better or worse on tasks summarising source code in different languages.

Our work also only focused on small Transformer models. While our models can be run on most commercially available workstations with little environmental impact, larger scale Transformers and LLMs present exciting opportunities for source code summarisation, which we have not investigated as part of this paper.

We also chose to evaluate our results against a substantial suite of traditional and LLM-based NLG metrics. While these metrics are all designed with the aim of complementing and being comparable to human expert evaluation, future work could be done to compare these metrics to human evaluation in the domain of source code summarisation.

8. Ethics Statement

The first ethical consideration of our research is the environmental impact of our research. We have taken steps to minimize this impact by choosing to training small models on commercially available workstation machines. Any future research into whether larger models are capable of outperforming the results we have achieved will have a larger environmental impact.

We also considered the dataset we have used. The data itself is comprised of publicly available Java source code, and the primary dataset we have used was compiled by LeClair and McMillan (2019). We also used data from the CodeSearchNet dataset (Husain et al., 2019), which is derived from open source projects on GitHub with licenses which permit the re-distribution of parts of code.

9. Bibliographical References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007, Online. Association for Computational Linguistics.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*.

Ayana, Shiqi Shen, Yu Zhao, Zhiyuan Liu, and Maosong Sun. 2016. Neural headline generation with sentence-wise optimization.

Satanjeev Banerjee and Alon Lavie. 2005. 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*, pages 65–72, Ann Arbor, Michigan. Association for Computational Linguistics.

Ondřej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Varvara Logacheva, Christof Monz, Matteo Negri, Aurélie Névéol, Mariana Neves, Martin Popel, Matt Post, Raphael Rubino, Carolina Scarton, Lucia Specia, Marco Turchi, Karin Verspoor, and Marcos Zampieri. 2016. Findings of the 2016 conference on machine translation. In *Proceedings of the First Conference on Machine Translation: Volume 2, Shared Task Papers*, pages 131–198, Berlin, Germany. Association for Computational Linguistics.

Michael Denkowski and Alon Lavie. 2014. Meteor universal: Language specific translation evaluation for any target language. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 376–380, Baltimore, Maryland, USA. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. 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)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Mahmoud El-Haj, Paul Rayson, and David Hall. 2014. Language independent evaluation of translation style and consistency: Comparing human and machine translations of camus' novel "the stranger". In *Text, Speech and Dialogue: 17th International Conference, TSD 2014, Brno, Czech Republic, September 8-12, 2014. Proceedings 17*, pages 116–124. Springer.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcode{bert}: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Sakib Haque, Aakash Bansal, and Collin McMillan. 2023. Label smoothing improves neural source code summarization. *arXiv preprint arXiv:2303.16178*.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred api knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 2269–2275. International Joint Conferences on Artificial Intelligence Organization.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Moussa Kamal Eddine, Guokan Shang, Antoine Tixier, and Michalis Vazirgiannis. 2022. FrugalScore: Learning cheaper, lighter and faster evaluation metrics for automatic text generation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1305–1318, Dublin, Ireland. Association for Computational Linguistics.

Alexander LeClair and Collin McMillan. 2019. 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)*, pages 3931–3937, Minneapolis, Minnesota. Association for Computational Linguistics.

Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online. Association for Computational Linguistics.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Chin-Yew Lin and Franz Josef Och. 2004. ORANGE: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 501–507, Geneva, Switzerland. COLING.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Junayed Mahmud, Fahim Faisal, Raihan Islam Arnob, Antonios Anastasopoulos, and Kevin Moran. 2021. Code to comment translation: A comparative study on model effectiveness & errors. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, pages 1–16, Online. Association for Computational Linguistics.

Mohammad Norouzi, Samy Bengio, Navdeep Jaitly, Mike Schuster, Yonghui Wu, Dale Schuurmans, et al. 2016. Reward augmented maximum likelihood for neural structured prediction. *Advances In Neural Information Processing Systems*, 29.

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

Jesse Phillips, David Bowes, Mahmoud El-Haj, and Tracy Hall. 2022. Improved evaluation of automatic source code summarisation. In *Proceedings of the 2nd Workshop on Natural Language Generation, Evaluation, and Metrics (GEM)*, pages 326–335, Abu Dhabi, United Arab Emirates (Hybrid). Association for Computational Linguistics.

Maja Popović and Hermann Ney. 2007. Word error rates: Decomposition over POS classes and applications for error analysis. In *Proceedings of the Second Workshop on Statistical Machine Translation*, pages 48–55, Prague, Czech Republic. Association for Computational Linguistics.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551.

Shiqi Shen, Yong Cheng, Zhongjun He, Wei He, Hua Wu, Maosong Sun, and Yang Liu. 2016. Minimum risk training for neural machine translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1683–1692, Berlin, Germany. Association for Computational Linguistics.

Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the evaluation of neural code summarization. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1597–1608.

Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. 2020. A human study of comprehension and code summarization. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 2–13.

Danny van Bruggen, Federico Tomassetti, Roger Howell, Malte Langkabel, Nicholas Smith, Artur Bosch, Malte Skoruppa, Cruz Maximilien, ThLeu, Panayiotis, Sebastian Kirsch, Simon, Johann Beleites, Wim Tibackx, jean pierre L, André Rouél, edefazio, Daan Schipper, Mathiponds, Why you want to know, Ryan Beckett, ptitjes, kotari4u, Marvin Wyrich, Ricardo Morais, Maarten Coene, bresai, Implex1v, and Bernhard Haumacher. 2020. javaparser/javaparser: Release javaparser-parent-3.16.1.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with BERT. *CoRR*, abs/1904.09675.

Wei Zhao, Maxime Peyrard, Fei Liu, Yang Gao, Christian M. Meyer, and Steffen Eger. 2019. MoverScore: Text generation evaluating with contextualized embeddings and earth mover distance. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 563–578, Hong Kong, China. Association for Computational Linguistics.

Ziyi Zhou, Huiqun Yu, Guisheng Fan, Zijie Huang, and Kang Yang. 2023. Towards retrieval-based neural code summarization: A meta-learning approach. *IEEE Transactions on Software Engineering*.

A. CodeSumBART Example Predictions

```
Selected Summaries (5 longest, 5 shortest, 5 mean.)

Shortest Summaries longer than 2 tokens:
        Source: public hashtable get hash ( ) { return attributes ; }
        Prediction: returns the entireable of contains guaranteed to
           filter the attribute. this .
        Reference: returns the hashtable that is used to store the
           attributes of this object
        WER: 0.615

        Source: public void close ( ) { _ disconnect ( ) ; }
        Prediction: closeoses the connectionagramrovider. creates
           connection  the chatacle thread.
        Reference: closes the dataprovider and the connection to the
           oracle database
        WER: 0.7

        Source: public responses get addressing responses ( ) { return
            addressing responses ; }
        Prediction: getss value of addressing to to addressing
        Reference: return the type of responses required by addressing
           .
        WER: 0.875

        Source: public int get colon pos ( ) { return colon pos ; }
        Prediction: gets position of   code   token token token  or  1
            if not present
        Reference: returns position of   code     token  or  1 if not
           present.
        WER: 0.4

        Source: public chord node get successor ( ) { return this .
           successor ; }
        Prediction: returns the successor of this chord.
        Reference: returns the successor of this peer.
        WER: 0.167


Longest Summaries:
        Source: public void test clg07 ( ) throws exception { assert
           equals ( 0 , test utilities . bool search ( " ( cccc . cc .
            cccn ) . n . c " , " ccccc . cccn " ) ) ; assert equals (
           0 , test utilities . bool search ( " ( cl . cccc . cc .
           ccccn ) . n . c " , " ccccc . cccn " ) ) ; assert equals (
           1 , test utilities . bool search ( " ( cccc . cc ) . ( cccn
            ) . n . c " , " ccccc . cccn " ) ) ; assert equals ( 0 ,
           test utilities . bool search ( " ( cc br . ccn ) . ( occ )
           " , " br ccccc . cccn . occc " ) ) ; assert equals ( 1 ,
           test utilities . bool search ( " ( cc br ) . ( ccn ) . (
           occ ) " , " br ccccc . cccn . occc " ) ) ; assert equals (
           1 , test utilities . bool search ( " ( cc [ br , cl ] ) . (
            ccn ) . ( occ ) " , " br ccccc . cccn . occc " ) ) ; }
        Prediction: finds the virtualpoint for the reference reference
            the reference reference to
        Reference: returns a virtual point on the line between the
           point closest geographically to
        WER: 0.769

        Source: public void test clg07 ( ) throws exception { assert
           equals ( 0 , test utilities . bool search ( " ( cccc . cc .
            cccn ) . n . c " , " ccccc . cccn " ) ) ; assert equals (
```

0 , test utilities . bool search (" (cl . cccc . cc . ccccn) . n . c " , " ccccc . cccn ")) ; assert equals (1 , test utilities . bool search (" (cccc . cc) . (cccn) . n . c " , " ccccc . cccn ")) ; assert equals (0 , test utilities . bool search (" (cc br . ccn) . (occ) " , " br ccccc . cccn . occc ")) ; assert equals (1 , test utilities . bool search (" (cc br) . (ccn) . (occ) " , " br ccccc . cccn . occc ")) ; assert equals (1 , test utilities . bool search (" (cc [br , cl]) . (ccn) . (occ) " , " br ccccc . cccn . occc ")) ; }
Prediction: sets the the check the the class is not if that the
Reference: set how to compare to this conditionfactor. value is true implies match for
WER: 0.923

Source: public void test clg07 ( ) throws exception { assert equals (0 , test utilities . bool search (" (cccc . cc . cccn) . n . c " , " ccccc . cccn ")) ; assert equals (0 , test utilities . bool search (" (cl . cccc . cc . ccccn) . n . c " , " ccccc . cccn ")) ; assert equals (1 , test utilities . bool search (" (cccc . cc) . (cccn) . n . c " , " ccccc . cccn ")) ; assert equals (0 , test utilities . bool search (" (cc br . ccn) . (occ) " , " br ccccc . cccn . occc ")) ; assert equals (1 , test utilities . bool search (" (cc br) . (ccn) . (occ) " , " br ccccc . cccn . occc ")) ; assert equals (1 , test utilities . bool search (" (cc [br , cl]) . (ccn) . (occ) " , " br ccccc . cccn . occc ")) ; }
Prediction: constructbometricometric cumulative chart cumulative option
Reference: hypergeometric bar chart with cumulative option
WER: 0.5

Source: public void test clg07 ( ) throws exception { assert equals (0 , test utilities . bool search (" (cccc . cc . cccn) . n . c " , " ccccc . cccn ")) ; assert equals (0 , test utilities . bool search (" (cl . cccc . cc . ccccn) . n . c " , " ccccc . cccn ")) ; assert equals (1 , test utilities . bool search (" (cccc . cc) . (cccn) . n . c " , " ccccc . cccn ")) ; assert equals (0 , test utilities . bool search (" (cc br . ccn) . (occ) " , " br ccccc . cccn . occc ")) ; assert equals (1 , test utilities . bool search (" (cc br) . (ccn) . (occ) " , " br ccccc . cccn . occc ")) ; assert equals (1 , test utilities . bool search (" (cc [br , cl]) . (ccn) . (occ) " , " br ccccc . cccn . occc ")) ; }
Prediction: test test checks fail a xpath elements returned returns fail x
Reference: this test will perform an xpath query which will return
WER: 0.9

Source: public void test clg07 ( ) throws exception { assert equals (0 , test utilities . bool search (" (cccc . cc . cccn) . n . c " , " ccccc . cccn ")) ; assert equals (0 , test utilities . bool search (" (cl . cccc . cc . ccccn) . n . c " , " ccccc . cccn ")) ; assert equals (1 , test utilities . bool search (" (cccc . cc) . (cccn) . n . c " , " ccccc . cccn ")) ; assert equals (0 , test utilities . bool search (" (cc br . ccn) . (occ) " , " br ccccc . cccn . occc ")) ; assert equals (1 , test utilities . bool search (" (cc br) . (ccn) . (

```
      occ ) " , " br ccccc . cccn . occc " ) ) ; assert equals (
      1 , test utilities . bool search ( " ( cc [ br , cl ] ) . (
       ccn ) . ( occ ) " , " br ccccc . cccn . occc " ) ) ; }
   Prediction: set the line. to draw origin shape.
   Reference: sets the line used to label this series.
   WER: 0.75


Mean Summaries:
   Source: private void fire waypoints available ( gps unit event
       evt ) { for ( iterator it = _ listeners . iterator ( ) ;
      it . has next ( ) ; ) { gps unit event listener l = ( gps
      unit event listener ) it . next ( ) ; l . waypoints
      available ( evt ) ; } }
   Prediction: resetets all properties to their. for the.
      requests
   Reference: resets all fields to values valid for validation.
   WER: 0.75

   Source: public void test assign graph pool ( ) { o data
      manager . assign graph pool ( ) ; assert true ( o data
      manager . o dex . is open ( ) & & o data manager . o graph
      pool . is open ( ) ) ; o data manager . close db ( ) ; }
   Prediction: sets the bindings are not files types are be .
   Reference: whether internal bindings or and external binding
      should be used.
   WER: 0.8

   Source: public int get int ( string key ) { int i = 0 ; try {
      i = integer . parse int ( props . get property ( key ) ) ;
      } catch ( throwable t ) { logger . log ( level . warning ,
      " could not parse integer value " , t ) ; } return i ; }
   Prediction: sets the audio renderer. use  this of these
   Reference: set the audio renderer to use. one of
   WER: 0.75

   Source: public void work on ( assembly a ) { composite node
      new node = new composite node ( name ) ; for ( int i = 0 ;
      i < number nodes ; i + + ) { component node node = (
      component node ) a . pop ( ) ; new node . insert ( node ) ;
       } a . push ( new node ) ; }
   Prediction: getss filterconfig. for this filter.
   Reference: return the filter configuration object for this
      filter.
   WER: 0.625

   Source: public void set active ( final boolean active ) { if (
       ( mode ! = mode . server ) & & ( ! in applet ) ) { if (
      active ) { status . set sort mode ( sort mode . remote , "
      remote " ) ; } else { status . set sort mode ( sort mode .
      no _ sort , " no sort " ) ; } } }
   Prediction: inv be be called for
   Reference: must not be called.
   WER: 1.0


Mean Word Error Rate: 0.566
Mean Word Error Rate (Shortest 100 summaries): 0.520
Mean Word Error Rate (Mean 100 summaries): 0.521
Mean Word Error Rate (Longest 100 summaries): 0.562
```

B. Correlation for Evaluation Metrics

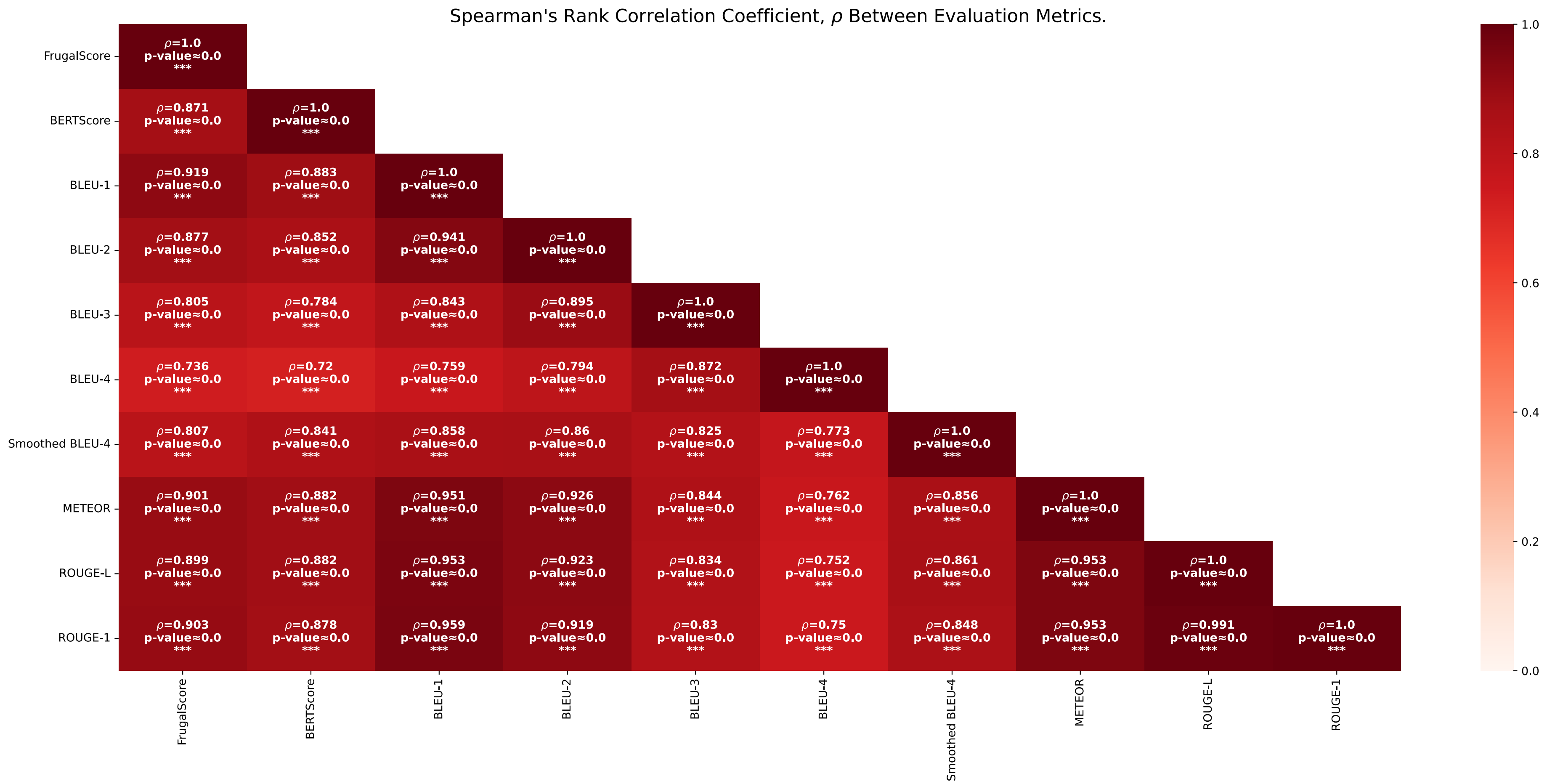


Figure 2: Spearman's Rank Correlation Coefficient, using 100% of the Evaluation Split

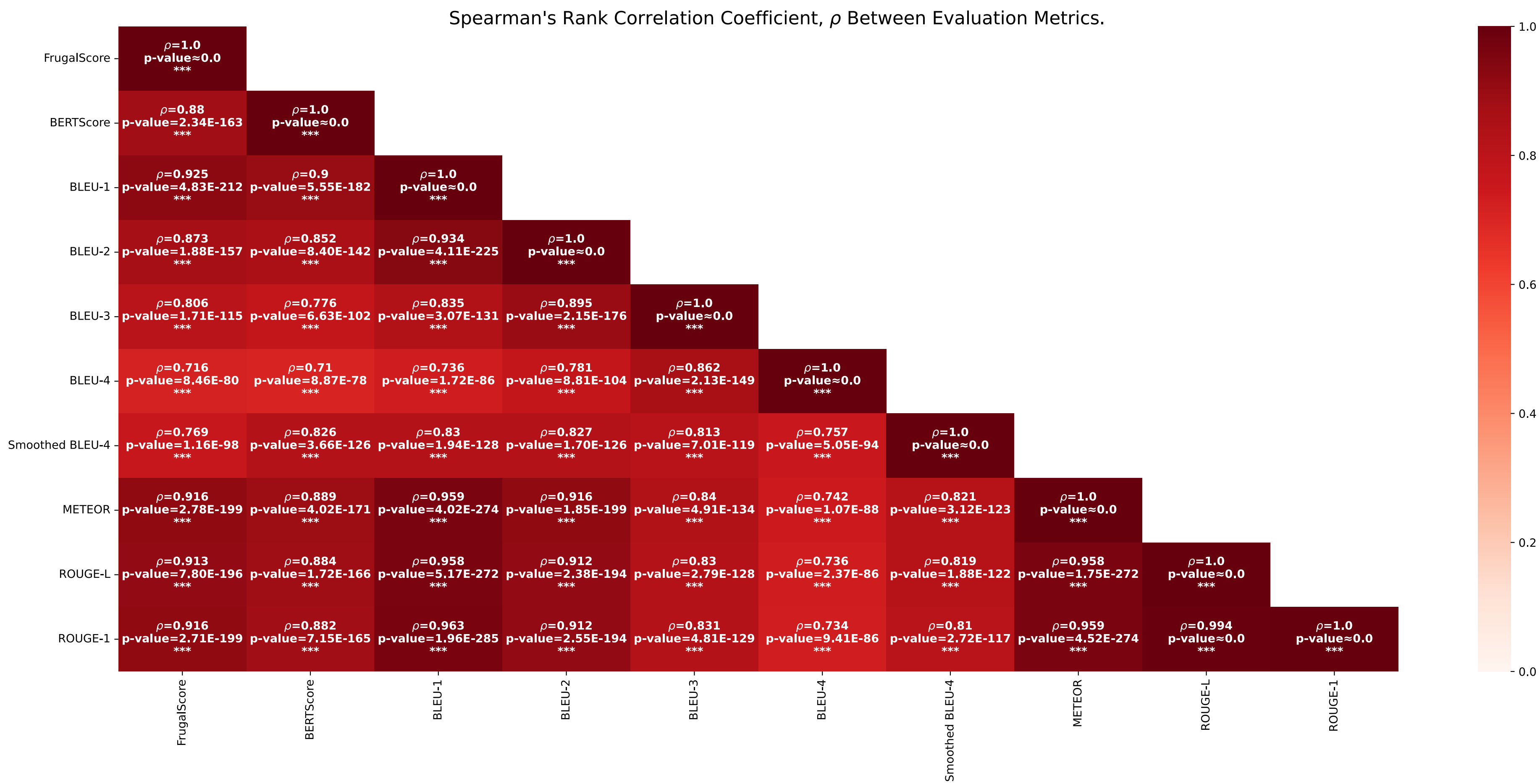


Figure 3: Spearman's Rank Correlation Coefficient, using 1% of the Evaluation Split