Improved Evaluation of Automatic Source Code Summarisation

Jesse Phillips and **David Bowes** and **Mahmoud El-Haj** and **Tracy Hall**
School of Computing and Communications
Lancaster University, UK
{j.m.phillips, d.h.bowes, m.el-haj, tracy.hall}@lancaster.ac.uk

Abstract

Source code summarisation is a vital tool for the understanding and maintenance of source code as summarisations can be used to explain code in simple terms. However, source code with missing, incorrect, or outdated summaries is a common occurrence in production code. Automatic source code summarisation seeks to solve these issues by generating up-to-date summaries of source code methods. Recent work in automatically generating source code summaries uses neural networks for generating summaries; commonly Sequence-to-Sequence or Transformer models, pretrained on method-summary pairs. The most common method of evaluating the quality of these summaries is comparing the machine-generated summaries against human-written summaries. Summaries can be evaluated using n-gram-based translation metrics such as BLEU, METEOR, or ROUGE-L. However, these metrics alone can be unreliable and new Natural Language Generation metrics based on large pretrained language models provide an alternative. In this paper, we propose a method of improving the evaluation of a model by improving the preprocessing of the data used to train it, as well as proposing evaluating the model with a metric based off a language model, pretrained on a Natural Language (English) alongside traditional metrics. Our evaluation suggests our model has been improved by cleaning and preprocessing the data used in model training. The addition of a pretrained language model metric alongside traditional metrics shows that both produce results which can be used to evaluate neural source code summarisation.

1 Introduction

Research producing models for neural source code summarisation frequently uses metrics designed for translation and Natural Language Generation (NLG) tasks, such as BLEU (Papineni et al., 2002), METEOR (Banerjee and Lavie, 2005), (Denkowski and Lavie, 2014), and ROUGE-L (Lin, 2004). These metrics are oriented to tasks such as translation and summarisation. Mahmud et al. (2021) used these metrics and a shared dataset to compare state-of-the-art models for neural source code summarisation. These metrics are based on n-gram matching, which compares the lexical similarity of texts, but lacks a comparison of the meaning of the texts - the semantic similarity. A lack of semantic comparison means that machine-generated texts which share sequences of n-grams with reference texts score well regardless of their ability to convey a similar meaning; whereas texts which convey a similar meaning but don't share many sequences of n-grams with reference texts score poorly. For summarisation tasks, the ability to replicate the semantics of a text where the generated language may be abstractive is more significant than the ability to generate as many n-grams matching those in a human-written summary as possible because the purpose of a summary is to provide the reader with an overview of the meaning of a larger text (or in this case, a source code method).

NLG metrics based on Large Language Models (LLM) aim to improve the reliability of evaluation scores by capturing semantics through reliance on contextual embeddings. However, these metrics require large amounts of computational resources - making them expensive to run in comparison to traditional n-gram matching metrics. New efforts in making these LLM-based metrics more accessible attempt to reduce the number of parameters compared to previous LLM-based models whilst retaining similar accuracy. This allows for faster calculation of LLM-based metrics. One such new effort is the *FrugalScore* metric (Kamal Eddine et al., 2022) for evaluating NLG tasks. FrugalScore uses previous LLM-based metrics to train a miniature language model which learns on the internal mapping of the expensive metric. It is this model which is used for generating scores for pairs of sequences of text.

In this paper, our aim is to train a state-of-the-art neural network model (we use the NeuralCodeSum model (Ahmad et al., 2020)) designed for source code summarisation, using a dataset of source code - summary pairs as described in Section 2. The dataset we use is a modified version of the Funcom dataset (LeClair and McMillan, 2019). We selected this dataset to allow us to compare our model to previous research (Mahmud et al., 2021) which trains NeuralCodeSum on the same dataset and evaluates the results using traditional n-gram matching metrics. We then evaluate our trained model using the FrugalScore metric (Kamal Eddine et al., 2022) to show the ability of such a metric to evaluate neural source code summarisation.

1.1 Research Questions

RQ1.1 What does a comparison of commonly used metrics show about our model following our data cleaning?

We train our model and evaluate it using a series of n-gram-based NLG metrics. We can then use these results to compare to previous research.

RQ1.2 How does the model trained on our dataset compare against previous experiments on the same model?

We compare our results to previous research training NeuralCodeSum models. Comparing our model to Mahmud et al.'s (2021) model will show the effect of our improved pretraining of training data on improving the evaluation of the model.

RQ2 How do the results of traditional metrics compare to those of FrugalScore?

We present an alternative method of evaluating neural source code summarisation, by using a metric based on a language model - FrugalScore (Kamal Eddine et al., 2022). We show the difference between traditional and LLM-based metrics and the ability to use both for a more complete evaluation of source code summarisation.

2 Dataset

The data we used comes from the filtered version of the Funcom dataset (LeClair and McMillan, 2019). Funcom was proposed in the paper "Recommendations for Datasets for Source Code Summarization" as a dataset of Java methods with associated English Javadoc comments including summaries. There are three versions of the Funcom dataset: raw, filtered, and tokenised. The filtered dataset is chosen over the raw dataset as it removes automatically generated code and code without English summaries. We chose the filtered dataset over the tokenised dataset for the purposes of this experiment to give us greater control over the data preprocessing; the tokenised dataset implements their own removal of special characters, tokenisation, splitting of camel case, and lowercasing. These are all steps which we perform, but intend to control.

Mahmud et al. (2021) set out a method for preprocessing the Funcom dataset for use with NeuralCodeSum, alongside other neural networks for neural source code summarisation. However, this method does not require code to be compilable, and the script provided by the authors for replicating the experiment contains a flaw as described below (also, see Listing 1 and Appendix A).

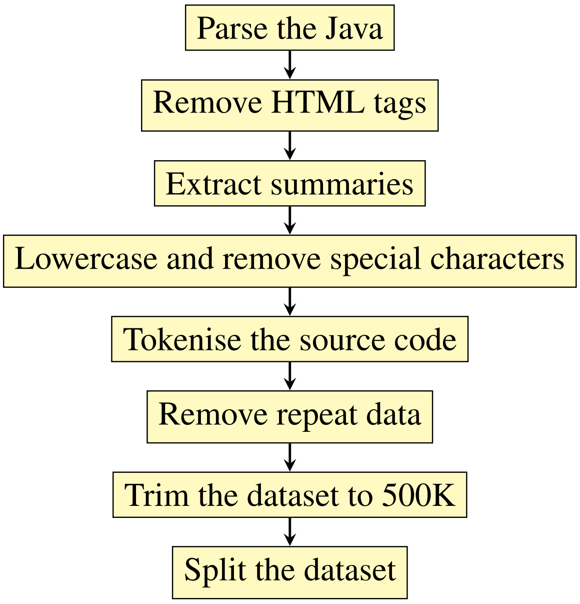
In attempting to remove comments, Mahmud et al.'s (2021) method strips all lines containing the string "//". Whilst this is the identifier for an inline comment in Java, the same string may be used elsewhere. For example: "//" occurs numerous times in our dataset as part of a URL.

Listing 1: Mahmud et al.'s (2021) method for removing comments: pseudocode

```
function remove_comments(method)
{
    create an array (of strings)
    for each line of text in a method:
    {
        if the line does not contain "//"
            add the line to the array
    }
    combine the contents of the array into a new method
    return the new method
}
```

We created a new method for preprocessing the filtered version of the Funcom dataset (LeClair and McMillan, 2019), based on that followed by Mahmud et al. (2021), with some changes to fix these issues. The first of these changes is parsing the Java code used to ensure that only compilable code is included in the experiment. We did this using JavaParser (van Bruggen et al., 2020). The use of only

Figure 1: Dataset Preprocessing



compilable code focuses the model on “production” code. Following the principle that we can disregard code which cannot be used in a real-world system to allow the model to focus solely on code used in production. We then used JavaParser to remove inline code comments written by the developers without the risk of damaging the code itself. Removal of natural language code comments from the code removes bias due to encountering potentially incorrect summaries written by the developers. Our method of using JavaParser to remove only strings the Java compiler would recognise as code comments ensures that potentially useful data within the Java source code which contains the string “//” does not get removed. For example:

```
// Example A:
public void play() {
    // If no sound file is there
       nothing can be played.
    ...

// Example B:
location = linkUrl.toString().
   replaceFirst("file:/", "file
   ://");
```

Example A would have the string “// If no sound file is there nothing can be played.” removed, but Example B would not be affected.

As Figure 1 shows, the dataset is first parsed by JavaParser, and method-comment pairs where the methods cannot be interpreted are removed from the data. During this parsing phase, comments within the methods are stripped from the methods to remove noise as described above.

HTML data is then removed from the comments, as remnants of HTML tags in the comments used for training may influence the predictions and skew the results of the experiment. This is because after removing special characters, leftover text from the HTML tags could be present in the training data, and the model would - therefore - attempt to replicate that pattern in evaluation.

Following this, we extract the section of each comment in a method-comment pair most likely to represent a basic summary. For example, from:

```
/**
 * Returns the pushes lowerbound
      of this board position.
 *
 * @return the pushes lowerbound
*/
```

we extract the string “Returns the pushes lowerbound of this board position.”. This is done by extracting the first line of non-whitespace (that is: not composed entirely of space and/or tab characters) text with more than 8 characters. As the comments in the method-comment pairs are extracted from method-level Javadoc, the first line of meaningful text in our dataset is usually a method summary.

These summaries are lowercased and special characters (characters which are not alphanumeric, full-stops, or apostrophes) are removed from them, in order to ensure each method has a plain natural language summary with minimal noise which could interfere with the neural network model. Much like with the removal of HTML tags, these special characters which may not add to the natural language summary of the method would be replicated by the model, potentially worsening results.

The source code methods are then tokenised. As part of our tokenisation, camel case phrases are split into individual words, punctuation is spaced out from words, and the text is then lowercased. This maintains the structure of the source code (including any structural information), whilst removing information which may be misleading. For example, one camel case word may contain a string of words which provide useful information to the model when tokenised.

Repeat data is then removed to prevent any bias on the final results. In our initial testing, we found our model scored highly on all metrics, but when we generated a histogram of the results, we discovered that this was due to an incredibly high number of perfect matches, rather than a single right-skewed peak. By comparing our test data to our training data, we discovered multiple copies of the same method in the dataset, which were present in both testing and evaluation data. Removing these repeats allows us to reduce this effect.

The size of the dataset is then reduced in order to decrease the time and compute power needed to run this experiment. The first 500,000 valid method-comment pairs are used as the data for this experiment. Selecting only the first 500,000 pairs is a technique used by Mahmud et al. (2021), which means that our results are still comparable. However, this means there is potential to improve the model further in future by training it on a larger dataset.

We further split our dataset into training, validation, and evaluation (80% / 10% / 10%) datasets. We chose this split as that is the same split chosen by both Ahmad et al. (2020) and Mahmud et al. (2021) for training and evaluating their NeuralCodeSum models. The data is split with a randomised mixture of code from multiple projects in each dataset to prevent the artificial inflation of results due to any one dataset having a majority of code from a single project within the larger dataset, which may cause artificial inflation of performance due to data snooping in our evaluation which wouldn't reflect real conditions (as suggested by both LeClair and McMillan (2019), and Mahmud et al. (2021)).

3 Research Methodology

We began by building our dataset, as described in the Section 2. We used the Funcom dataset (LeClair and McMillan, 2019), as this will allow us to compare our results to Mahmud et al.'s (2021) evaluation of NeuralCodeSum, and added our own preprocessing steps in order to improve our model. The model was trained and evaluated on a machine using an `Intel i9-12900KF` CPU, 128GB DDR4 RAM, and an `Nvidia GeForce RTX 3090` GPU, using the official implementation of NeuralCodeSum, `PyTorch 1.3`, and `Python 3.6`. The result data from the evaluation process is collected in JSON format. The code used to process the dataset is available at github.com/phillijm/JavaDatasetCleaner.

3.1 Methodology for RQ1.1

In testing our model using metrics, we selected the metrics previously used for research on NeuralCodeSum, Ahmad et al. (2020) and Mahmud et al. (2021) both used Smoothed BLEU-4 (Papineni et al., 2002), METEOR (Banerjee and Lavie, 2005), and ROUGE-L (Lin, 2004) for the evaluation of their NeuralCodeSum models. In using these metrics, we can compare our results to previous research more accurately. We also calculated BLEU-1-4. By analysing the difference between these, we can identify some strengths and weaknesses of our model. The smoothing technique used for Smoothed BLEU-4 was Lin and Och's (2004) technique. METEOR was measured using the official Java implementation of METEOR 1.5 (Denkowski and Lavie, 2014). ROUGE-L was measured using the Google Research python implementation of the metric (Liu, 2022).

3.2 Methodology for RQ1.2

To establish whether our model has improved based on our improvements to the preprocessing of training data, we compare our results from RQ1.1 to those of Mahmud et al. (2021) in Table 2. Mahmud et al. (2021) also trained a NeuralCodeSum Model on the Funcom dataset, but with a simpler approach to preprocessing. We hypothesise our model should outperform Mahmud et al.'s (2021) model in evaluation against the same metrics: Smoothed BLEU-4 (Papineni et al., 2002), METEOR (Banerjee and Lavie, 2005), and ROUGE-L (Lin, 2004), due to the improvements made in preprocessing the dataset used to train the model, as detailed in the Dataset Section, where we train the model on only compilable code, with the training summaries heavily sanitised.

3.3 Methodology for RQ2

Once this baseline comparison of our model against Mahmud et al.'s (2021) model has been established, we evaluate our model against the FrugalScore (Kamal Eddine et al., 2022) metric, in order to compare FrugalScore against traditional n-gram-based metrics for a neural source code summarisation task. Our aim in evaluating the model against FrugalScore is to see a FrugalScore value which is comparable with traditional metrics and be able to compare the distribution of FrugalScore to traditional metrics on a histogram. This will suggest that the results produced by the model are able to be measured reliably using FrugalScore as a metric.

4 Result Analysis

Table 1: Comparison of n-gram based metrics against FrugalScore, measuring summaries generated by our NeuralCodeSum Model.

Metric	Score
BLEU-1	37.70
BLEU-2	27.03
BLEU-3	19.94
BLEU-4	14.67
Smoothed BLEU-4	29.17
METEOR	19.93
ROUGE-L	45.82
FrugalScore	65.76

In answer to RQ1.1: how our model compares against commonly used metrics; as seen in Table 1, our model scores well with lower n-gram BLEU metrics, but that decreases with higher numbers of n-grams. Lin and Och (2004) suggest that a drop in BLEU from lower to higher orders of n-grams could correlate a high degree of adequacy (the generated summary is still understandable), but a lower degree of fluency in the language generated. A lack of fluency in the text generated by our model would also explain the lower METEOR score. Banerjee and Lavie (2005) suggest that whilst higher order n-gram BLEU scores can be used as an indirect measure of grammatical well-formedness, METEOR directly measures this. If the model were trained on a larger dataset, we expect this would be improved.

Comparing our results to previous experiments (RQ1.2), Table 2 shows an improvement in both Smoothed BLEU-4 and ROUGE-L scores when compared to Mahmud et al. (2021). These metrics suggest that our model has been improved in its ability to generate source code summaries by the improvements in Figure 1 in the preprocessing of the dataset used, with an improvement in Smoothed BLEU-4 of 7.67 and an improvement in ROUGE-L of 12.11.

We compared the performance of our model to previous experiments using the Smoothed BLEU-4, ROUGE-L, and METEOR metrics - as these metrics were presented by both Ahmad et al. (2020) and Mahmud et al. (2021). A possible future work would be to test these models against a wider range of n-gram-based metrics and LLM-based metrics.

In answer to RQ2: when comparing traditional n-gram-based natural language generation metrics to FrugalScore (Kamal Eddine et al., 2022) for the task of neural source code summarisation, our model scored an average FrugalScore value of 65.76, as shown in Table 1. The distribution of these can be seen in Figure 2, and more clearly in Figures 3-6 in Appendix B. The distribution of FrugalScore is a bimodal distribution, with both peaks to the right of the graph, and most results between 40 and 80. These second peak represents where FrugalScore measures the machine-generated summary as being identical or near-identical to the original human-written summary, which could potentially show some degree of overfitting in the model.

It is notable that - when compared to traditional metrics - FrugalScore gives far fewer "bad" results (for example, 31.308% of Smoothed BLEU-4 results were above 30, compared to 99.998% of FrugalScore results). FrugalScore also gives fewer "perfect" or "near-perfect" results than traditional metrics (for example, 6.571% of Smoothed BLEU-4 results were above 99, compared to 4.215% of FrugalScore results). The difference between these results suggests that FrugalScore gives more credit to summaries which other metrics rank poorly and less credit to summaries which other metrics rank highly, with higher median and mean values. This should be taken into account when directly comparing FrugalScore to traditional metrics.

Our results show that - on average - FrugalScore ranks summaries more highly than traditional metrics. This is likely either due to the ability of LLM-based metrics to take the semantics of a sentence into account, where n-gram-based metrics do not, or due to a possible overestimation of results. To determine this would require further human evaluation, as mentioned in the Limitations Section.

5 Related Work

5.1 Neural Source Code Summarisation Using Transformer Models

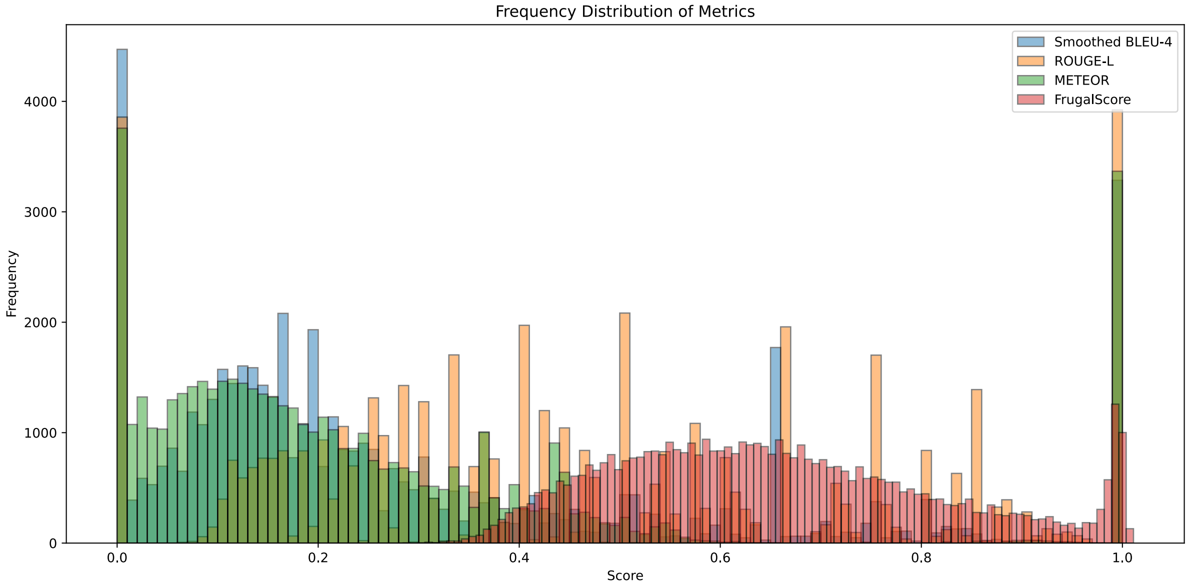
The Transformer model was initially proposed by Vaswani (Vaswani et al., 2017) and initially tested on the WMT 2014 English-to-German translation task. CodeBERT (Feng et al., 2020) and NeuralCodeSum (Ahmad et al., 2020) both use the Transformer architecture to form a model for neural source code summarisation, with NeuralCodeSum being a simple Transformer model, and CodeBERT being a larger bidirectional model, built on BERT (Devlin et al., 2019) and RoBERTa (Liu et al.,

Table 2: Comparison of metrics against those reported by previous papers.

Model	Smoothed BLEU-4	METEOR	ROUGE-L
Ahmad et al. (2020)*	44.58	26.43	54.76
Mahmud et al. (2021)	21.50	**27.78**	33.71
Our Model	**29.17**	19.93	**45.82**

* Ahmad et al.'s (2020) original experiment used a different dataset for training, so there is little relevance in directly comparing results.

Figure 2: Frequency Distribution of Metrics



2019).

Mahmud et al. (Mahmud et al., 2021) compare both of these models, as well as the Code2Seq (Alon et al., 2019) model when trained on the Funcom dataset (LeClair and McMillan, 2019).

5.2 Metrics for Evaluating Automatic Natural Language Generation Systems

Whilst there are many metrics for evaluating natural language generation tasks BLEU (Papineni et al., 2002) has become a common metric, with other metrics, such as ROUGE (Lin, 2004) and METEOR (Banerjee and Lavie, 2005) designed to be used alongside it, addressing potential flaws in BLEU itself. METEOR has been updated multiple times, with the current version being METEOR 1.5 (Denkowski and Lavie, 2014). Other metrics, such as ORANGE (Lin and Och, 2004) also suggest techniques which can be used to improve BLEU, such as the application of smoothing techniques.

Recent work in using language models as metrics to evaluate natural language generation tasks have presented a potential leap forwards in our ability to automatically evaluate such models. BERTScore (Zhang et al., 2020) is one such metric, as is MoverScore (Zhao et al., 2019). These metrics use large language models which require vast amounts of compute power, and may lead to ethical concerns due to the impact of training large language models on the environment. FrugalScore (Kamal Eddine et al., 2022) aims to go some way to solving this dilemma by reducing the environmental impact of the metric, whilst maintaining accuracy.

6 Conclusion

In testing a NeuralCodeSum model trained on a dataset of source code which has been parsed and had developer comments accurately removed, with accurate tokenisation of both source code and summaries, we have demonstrated the effect of ensuring a higher quality of training data has on improving the quality of a model - with our model outperforming that of Mahmud et al. (2021). In evaluating our model with FrugalScore alongside traditional metrics, we have shown how the two can be used alongside each other to provide an improved method of evaluating a model for neural source code summarisation.

Limitations

The first limitation to our research is that we have only tested the summarisation of Java source code in English. Whilst this research was limited in this aspect, it opens the possibility for future research, not only in the evaluation of neural source code summarisation, but also cross-language summarisation in general.

In this study, we used FrugalScore as a metric using a language model. Other metrics, such as BERTScore (Zhang et al., 2020) could be applied to compare language model-based metrics for this task, but this would require a far larger amount of GPU resources, as FrugalScore is designed as a lightweight metric. For this reason, we chose to use FrugalScore instead of other LLM-based metrics. Generating FrugalScore for our outputs took over 2 days and 18 hours using the HuggingFace implementation of the metric. Using more robust large language models also has a larger impact on ethics as the effect on the environment of these large language models would be greater. The use of further LLM-based metrics (potentially on smaller samples of data to reduce environmental impact and processing time) in an effort to show how these metrics compare to both n-gram-based metrics and human evaluation of neural source code summarisation is possible future work to expand upon this research.

The use of FrugalScore showed the possibility for the use of LLM-based metrics in analysing neural source code summarisation. The difference in distribution between FrugalScore and traditional metrics suggests that further analysis is needed to compare FrugalScore and traditional metrics with human evaluation. Only then can we determine whether FrugalScore better aligns with human evaluation or overestimates the quality of summaries.

This study also focused on the NeuralCodeSum model (Ahmad et al., 2020), as one example of a cutting edge model. However, other models, such as Code2Seq (Alon et al., 2019), or CodeBERT (Feng et al., 2020) have the potential to yield different results, something which could be explored in future.

The use of NeuralCodeSum also limited us in that to build the official implementation of the model required us to use old versions of Python (`Python 3.6`) and PyTorch (`PyTorch 1.3`), which are now deprecated. As time passes, the use of deprecated systems will produce an increased limitation on the reproducibility of our results.

Ethics Statement

The primary ethical considerations of our research are twofold: the environmental impact of our research, and the use of a large dataset of code we have not generated.

The dataset comes from LeClair and McMillan (2019) and consists of methods and Javadoc comments from publicly available Java source code.

Whilst in our research, we have taken precautions to limit our environmental impact (the selection of FrugalScore as a language model-based metric due to its lower environmental impact compared to BERTScore or MoverScore, and the selection of NeuralCodeSum as our test model, rather than a larger model with more parameters, such as CodeBERT), any research involving the training and evaluation of neural networks will have an environmental impact.

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. 2019. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*.

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.

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.

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.

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.

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.

Peter Liu. 2022. rouge-score 0.1.2.

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.

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.

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.

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, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. Bertscore: Evaluating text generation with bert. In *International Conference on Learning Representations*.

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.

A Appendix A

Listing 2: Mahmud et al.'s (2021) method for removing comments: official python implementation

```python
def remove_comments_inside_mthd(
    mthd: str) -> str:
    lines = mthd.split("\n")
    eachLine = []
    for line in lines:
        if "//" in line:
            continue
        else:
            eachLine.append(line)

    return "\n".join(eachLine)
```

B Appendix B

Below, we have provided histograms showing 1-to-1 comparisons of the frequency distribution of the FrugalScore metric against BLEU-1, Smoothed BLEU-4, METEOR, and ROUGE-L.

Figure 3: Frequency Distribution of BLEU-1 vs FrugalScore

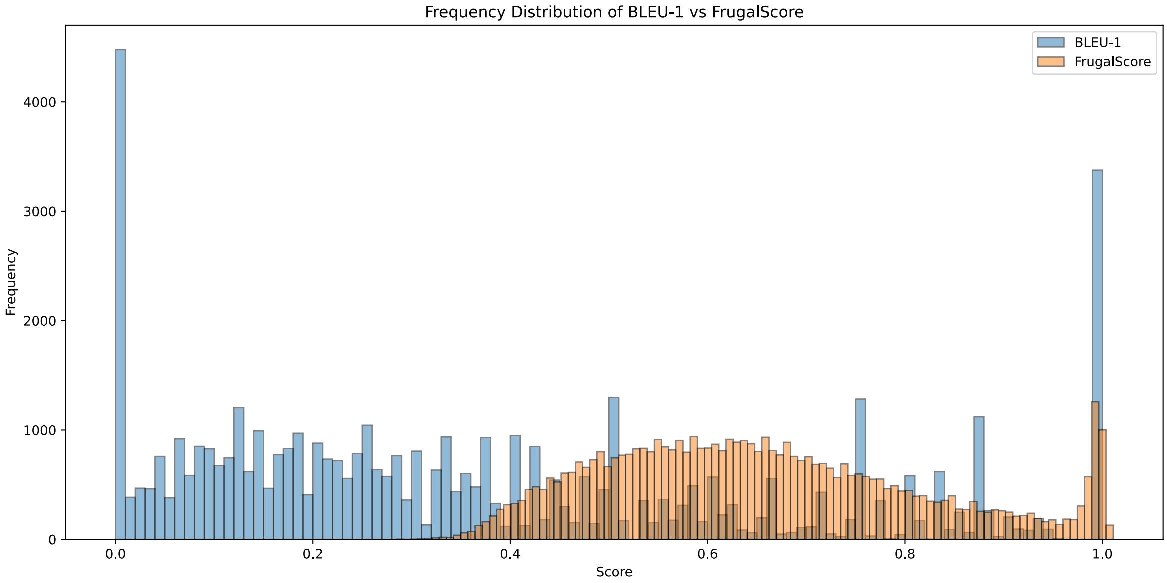


Figure 4: Frequency Distribution of Smoothed BLEU-4 vs FrugalScore

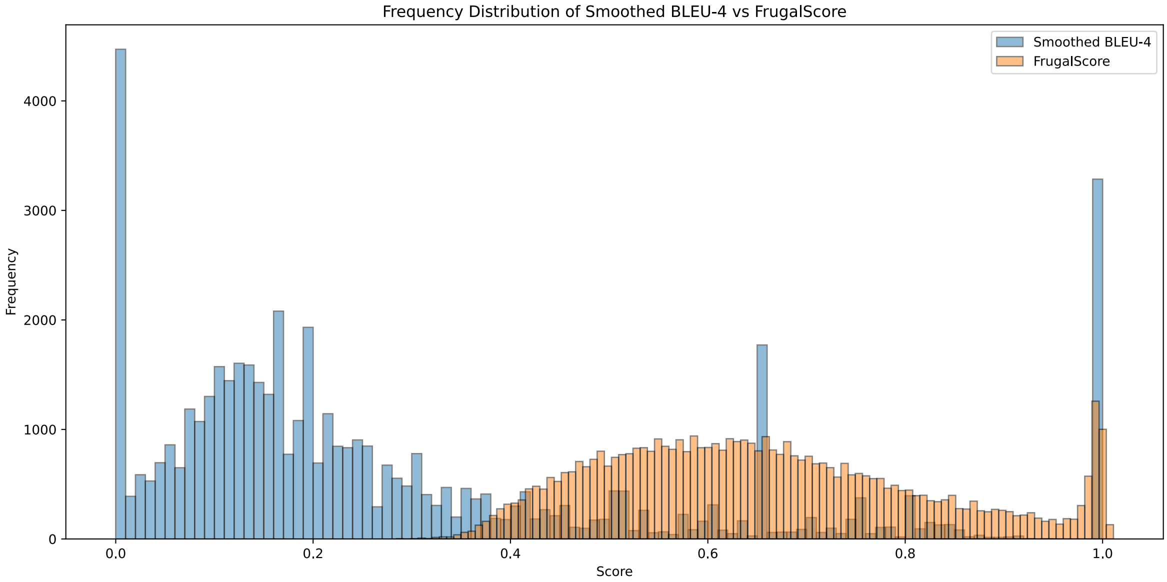


Figure 5: Frequency Distribution of METEOR vs FrugalScore

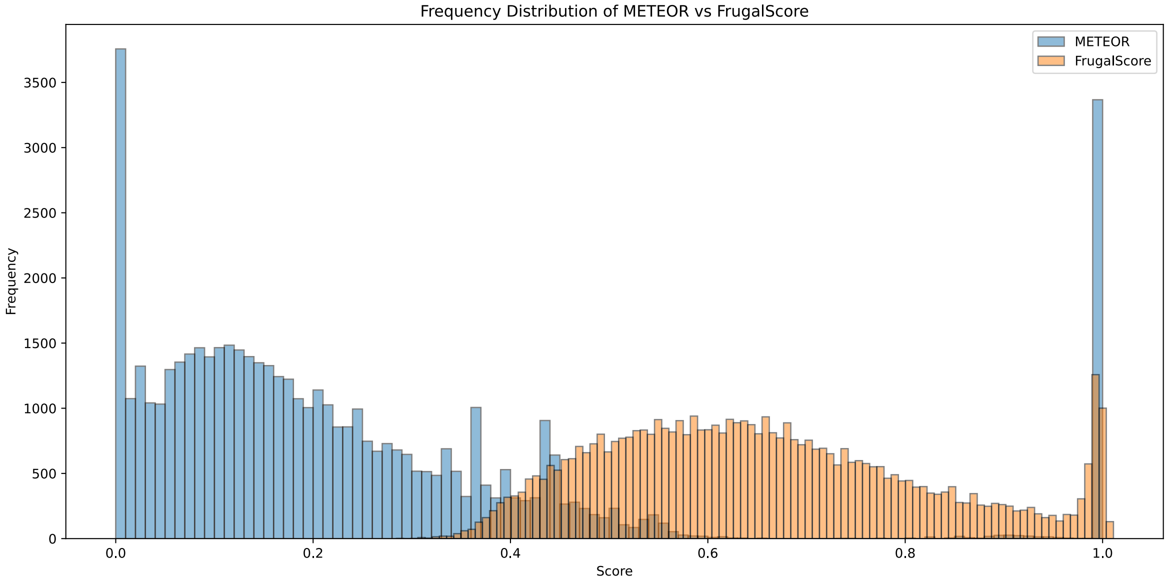
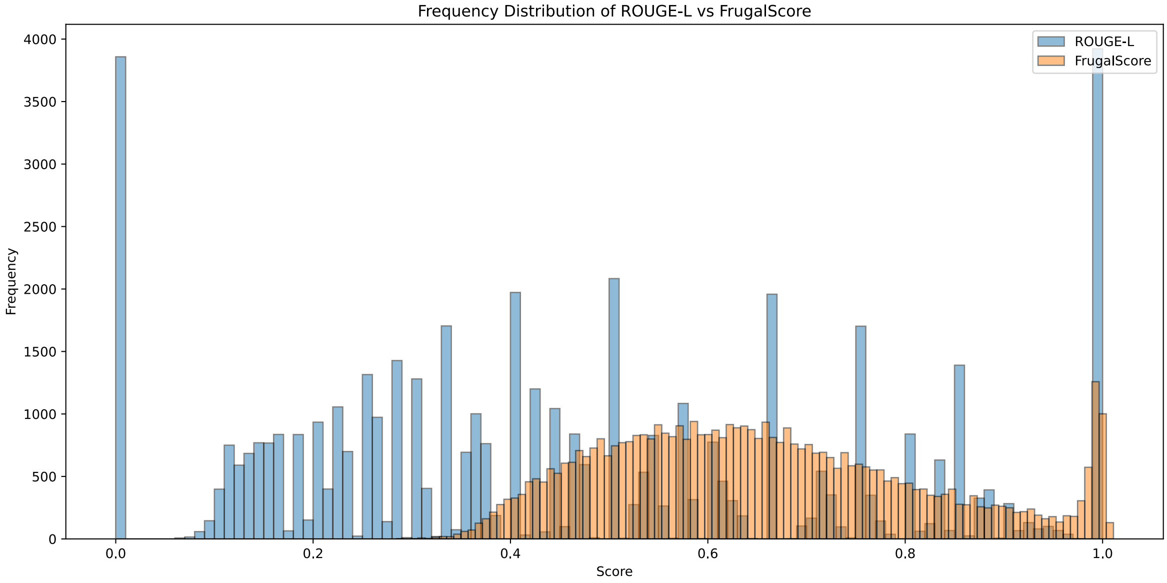


Figure 6: Frequency Distribution of ROUGE-L vs FrugalScore


Frequency Distribution of ROUGE-L vs FrugalScore
ROUGE-L
FrugalScore
Frequency
Score