Ensemble Models for Neural Source Code Summarization of Subroutines

Alexander LeClair, Aakash Bansal, and Collin McMillan
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA
{aleclair, abansal1, cmc}@nd.edu

***Abstract*—A source code summary of a subroutine is a brief description of that subroutine. Summaries underpin a majority of documentation consumed by programmers, such as the method summaries in JavaDocs. Source code summarization is the task of writing these summaries. At present, most state-of-the-art approaches for code summarization are neural network-based solutions akin to seq2seq, graph2seq, and other encoder-decoder architectures. The input to the encoder is source code, while the decoder helps predict the natural language summary. While these models tend to be similar in structure, evidence is emerging that different models make different contributions to prediction quality – differences in model performance are orthogonal and complementary rather than uniform over the entire dataset. In this paper, we explore the orthogonal nature of different neural code summarization approaches and propose ensemble models to exploit this orthogonality for better overall performance. We demonstrate that a simple ensemble strategy boosts performance by up to 14.8%, and provide an explanation for this boost. The takeaway from this work is that a relatively small change to the inference procedure in most neural code summarization techniques leads to outsized improvements in prediction quality.**

***Index Terms*—source code summarization, automatic documentation generation, neural networks**

I. Introduction

A source code "summary" of a subroutine is a natural language description of that subroutine. Summaries are the foundation of many documentation systems such as JavaDocs, where the summary is used to help programmers quickly gain an understanding of the purpose of the subroutine, without actually reading its source code [1]. The task of writing these summaries has come to be known as source code summarization [2], and has been an active research area for decades because of a mismatch between programmer behavior and their expectations: Programmers tend to avoid writing source code summaries due to the time cost and manual effort [3], [4]. Yet at the same time, programmers rely on good documentation written by others [5]. The result that automated solutions to code summarization are a very high value target in software engineering research.

Intense research interest has lately been focused on neural source code summarization approaches. These approaches rely on datasets of millions of examples of code and code summaries (e.g. [6], [7]) to train a neural network model to predict a source code summary. Almost all neural approaches to code summarization are some form of an attentional encoder-decoder model, in which the encoder creates a vectorized representation of source code, and the decoder represents the natural language summary. In the past five years, neural approaches have almost completely superseded alternatives such as sentence templates or IR-based keyword extraction [8]–[10].

Current strategies to neural code summarization can be broadly classified by the type of information they focus on modeling in the encoder. There are approaches that treat code as text, focusing on the identifier names and other natural language content buried in code [11], [12]. There are approaches that encode the context of other code in the same source file [13]. Some techniques model the structure of source code with an RNN by linearizing structural representations such as the AST [12], [14], [15]. And there is very active scrutiny of GNN-based encoders to model structures such as the AST or CPG [16]–[18]. All of these lines of inquiry are showing promise and continue to advance the state of the art.

Recent work in source code summarization has been focusing on providing models with ever more complex representations of code with the assumption that it will yield better and better predictions of code summaries [13], [15], [19], [20], [20]. This approach mirrors progress in many other fields such as machine translation or image captioning [21], [22]. However, hints from prior work point to a complementary relationship among neural models of code summarization, rather than a competitive one. Different types of information modeled by the encoder seem to provide orthogonal improvements to the predictions. For example, LeClair *et al.* [15] pointed out at ICSE'19 that their flattened AST approach improved predictions for some Java methods in their dataset but not others – while they observed an improvement in aggregate BLEU scores, this improvement was concentrated in a subset of methods. They found a mixture of their approach and a text-only encoder approach to provide the best overall results. The point is that significant progress may be possible by combining encodings that provide orthogonal improvements.

Meanwhile, "ensemble" models have a long tradition in many research areas as a way to combine contributions from diverse data sources [23], [24]. Examples include improving cancer diagnosis accuracy [25], weather prediction [26], stock market classification [27], solar power output efficacy [28],

This work is supported in part by *redacted for blind review*.

and many others. Sagi *et al.* [24] point out that the reason ensemble models work is that they increase data source diversity, avoid overfitting, and defray problems related to models stuck at local minima. Yet the benefits ensemble models may offer to code summarization have not been thoroughly studied.

In this paper, we combine different, complementary source code encoders for summarization via ensemble models. Our paper has two parts: 1) we conduct an empirical study in which we compare off-the-shelf ensemble technique to combine several baseline code summarization models from related literature, and 2) we provide a rationale behind the increases we observe from the ensemble techniques, showing how different models make different contributions to prediction performance.

We release all code, datasets, and other scripts to help other researchers via our online appendix (see Section IX).

II. Background & Related Work

This section discusses the key supporting technologies and related work to this paper, namely source code summarization in SE research and ensemble models generally.

A. Source Code Summarization

Recent work in source code summarization is overwhelmingly centered around the use neural networks and deep learning architectures. The work can be broadly categorized into four research directions:

Text-Based: Text-Based models use the text of the source code itself as a sequence, relying on the words that appear in the code to generate a summary. These methods usually use a sequence-to-sequence-like model design. The input to these models is the source code being summarized. The intended output is the summary of that source code. These approaches were first described around 2016, such as by Iyer *et al.* [11]. These approaches are still frequently used as baselines against which to evaluate newer approaches.

Flat-Structure: Flat-Structure models take the structure of the code, usually in the form of the AST, and flatten it into a sequence. For example, using a depth-first traversal to generate a sequence from the tree. After the early successes of text-based neural approaches, a trend formed in which several papers described code summarization approaches based on this flattened AST structure. For example, Hu *et al.* create a technique for flattening the AST called structure-based traversal (SBT). Their technique retains the structural information of the AST during the flattening process by adding a series of brackets and braces to group AST nodes. Other, more standard tree traversal techniques such as pre-order or post-order are considered lossy, in that the original AST may not be able to be reconstructed from the flattened output. Alon *et al.* generated paths between the nodes of the AST as a way to flatten the structure. They randomly selected multiple pairs of nodes in the AST for each method, and use the path between nodes as a flattened representation.

LeClair *et al.* observed that the language used in the code and the structure of the code contains orthogonal information

	Text	Context	Flat	GNN
2016 Iyer *et al.* [11]	x			
2017 Loyola *et al.* [29]	x			
2017 Lu *et al.* [30]	x			
2018 Hu *et al.* [20]	x	x		
2018 Liang *et al.* [31]			x	
2018 Hu *et al.* [14]			x	
2019 LeClair *et al.* [15]	x		x	
2019 Alon *et al.* [32]			x	
2019 Fernandes *et al.* [33]				x
2020 LeClair *et al.* [16]	x			x
2020 Haque *et al.* [13]	x	x	x	
2020 Ahmad *et al.* [34]	x		x	
2021 Zügner *et al.* [18]	x	x	x	
2021 Liu *et al.* [17]		x		x
2021 Bansal *et al.* [35]	x	x	x	

Fig. 1: Comparison of recent source code summarization research categorized into four broad categories: Text-Based, Structure-Based, Flat-Structure, and GNN-Structure. These categories reflect the type of input data the models use for source code summarization.

and could remain separate inputs to the model. They adapted the SBT approach to what they call the SBT AST Only (SBT-AO), which removed all identifiers from the SBT representation. They then had two inputs, one for the source code sequence and one for the SBT-AO. This allowed the model to learn the information from how the source code is written from one input, while also learning only from the structure with the other input.

Context-Based: Context-Based models rely on information outside of the method or snippet such as API calls [20] or other methods in the project [13]. For example, Haque *et al.* [13] show how other methods from the same file can provide additional needed context for a method summary. In an example, they show a simple setter method "setIntermediate" which sets a value to a passed parameter. The comment for this function is "sets the intermediate value for this flight" but nowhere in the method does the word "flight" appear. Other methods in the file do contain the word flight, since the project in question has to do with getting flight information. Recently, Bansal *et al* [35] developed a project-context method that uses the project context in addition to the method tokens and file context. They created a set of embeddings for each level in the project-file-method hierarchy and provided the model with each representation.

GNN-Structure: GNN-Structure-based models retain structure information in graph or tree formats. For instance, LeClair *et al.* [16] build upon their earlier work with flattened structure models by using both a source sequence input and a GNN to learn AST node representations. They found that the AST was able to learn better structure representations than a flattened AST. Liu *et al.* [17] combine a retrieval based technique and a GNN generated summary to produce summaries. They aggregate summaries from similarly structured code along side a GNN to generate the summary of a method.

B. *Ensemble Models*

An ensemble model is one in which several other models are aggregated to generate a single output. Ensemble models are used in a variety of applications ranging from neural machine translation [36] to stock market prediction [37]. Ensemble models have been shown to reach state-of-the-art results in many different areas [24]. The goal of ensembling is to get a "best of all worlds" output, in which we can take advantage of a model's relative strengths while simultaneously decreasing the effect of its weaknesses. Ensemble models work by aggregating a collection of models trained for the same task.

There are a variety of aggregation techniques that are used to combine model outputs. One commonly used aggregation technique is to average the outputs of all the models in the ensemble. When applied to text generation this aggregation technique averages the softmax output of each model for every time step in the sequence. More sophisticated aggregation techniques can also be used, such as an SVM, neural network, or weighted sum. Aggregation techniques that use a learning algorithm are known as meta-learning techniques. By aggregating these different models we can take advantage of orthogonal output. For example, LeClair *et al.* showed that their non-AST and AST models learned to summarize orthogonal subsets of Java methods, leading them to test a simple ensemble. Their ensemble model outperformed both their non-AST and AST models and showed the potential viability of a more sophisticated ensemble approach.

When working with ensemble models there are three high level design concepts: 1) the data used to train each model, 2) the models used in the ensemble, and 3) the procedure for aggregating those models. When determining model input data, ensemble models are considered either dependent or independent. In a dependent ensemble, each model is dependent on the output from another model. An example of a dependent ensemble model would be AdaBoost [38], where subsequent models are trained on previously mis-classified training data. Independent ensembles use a collection of independently trained models. Two of the primary techniques used for independent ensembles is bagging and stacking. In bagging, each model is trained on a subset of the data, while in stacking each model is trained on the entire dataset. In both dependent and independent ensembles the outputs of each model is aggregated to generate a single prediction. In this paper we focus on independent ensemble techniques.

C. *Metrics*

BLEU [39] is an automated metric commonly used to score the output of text generation and translation models. BLEU scores are commonly used in source code summarization tasks to rate the summary quality of a model. The BLEU algorithm scores the overlap of N-Grams in a predicted and reference text. To achieve a final single score many researchers use an aggregate of BLEU-1 to BLEU-4 which will count the number of overlaps in 1,2,3,4-Grams of the predicted text in the reference text.

III. RESEARCH QUESTIONS

The research objective of this paper is to study the effects of ensemble models for neural source code summarization, and to determine the orthogonality of different models that contribute to the ensembles. We ask the following two research questions:

RQ$_1$ What is the performance difference in terms of aggregate BLEU scores of existing baselines, when combined using a simple aggregating procedure?

RQ$_2$ What is the difference in the vector space representations of the functions in the models contributing to the ensembles?

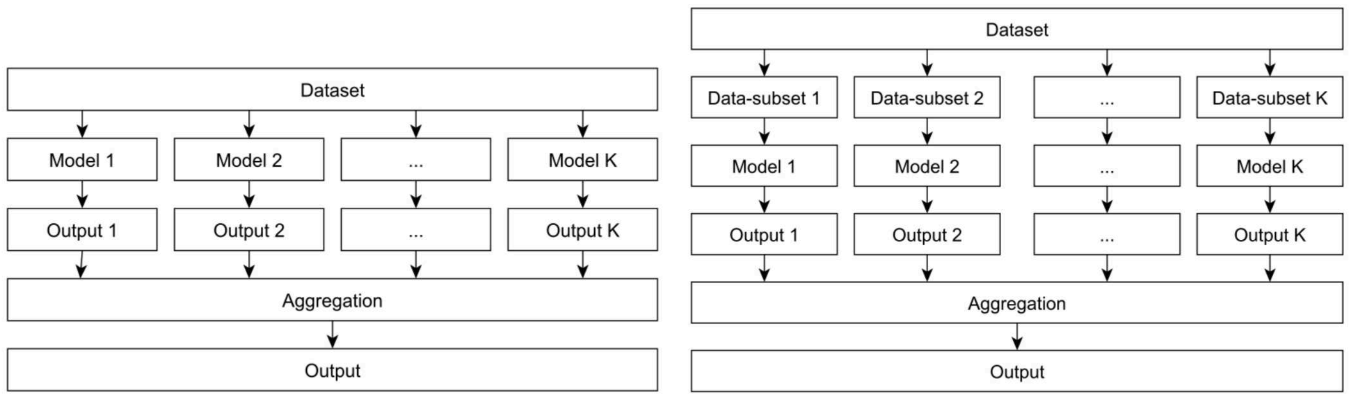
The rationale behind RQ$_1$ is that many models have been proposed for source code summarization, yet related literature does not describe how these models may contribute in an ensemble with other models. Our goal with this RQ is to cast a wide net and include many different model types, with many different source code input features, while keeping the aggregation procedure simple so that the results may be explained and more easily reproduced. To further focus on the model contribution, we use independent ensemble techniques. We use aggregate BLEU scores because that is by far the most common way in which neural source code summarization techniques are evaluated in related work.

The rationale behind RQ$_2$ is that each model may make orthogonal contributions to the predictions, yet some models may perform best because they provide the most orthogonal view of the source code. It is useful to know which models make the most different contributions because ensemble models work best by combining diverse inputs [24]. We focus on sets of subroutines for which each model performs the best.

IV. DATASETS

The data used in this work is the Java dataset released by LeClair *et al.* on recommendations for datasets for source code summarization [6]. This dataset contains 2.1 million code comment pairs in the Java programming language and is available in two formats, 1) filtered and 2) tokenized. The filtered version of the dataset has taken the raw data and filtered it down to 2.1 million code/comment pairs, but has not applied any additional processing to the text itself. The tokenized format has had text processing applied and is available in a vectorized format. We use the tokenized version of the dataset for this work because it has already been cleaned and processed based on the procedures outlined in other related works [13]–[15], [32].

We also utilize the adaptation of the Java dataset as outlined in the work by Haque *et al.*. This adaption adds a set of file context vectors for each method in the dataset. The file context consists of method vectors for each method that exists within the same file. This allows the model to learn additional context from the surrounding methods. We use this set to train the file-context (FC) versions of the models [13], more details can be found in Section V.



(a) (b)

Fig. 2: Ensemble architecture diagram for (a) stacking and (b) bagging techniques.

A. Threats to Validity

One threat to validity is the dataset we use. Our dataset contains only a single programming language (Java) and the ensemble models may not generalize to other programming languages. While datasets in other languages are available, few are as large and complete as the Java dataset provided by LeClair *et al.* and Haque *et al.* [6], [13] which contains around 2.1 million examples of source code, comments, ASTs as sequences and trees, and file context vectors. Using this dataset allowed us to use six model architectures from the literature as both baselines and component models for our ensemble techniques.

Ensembling models has a combinatorial explosion affect on the number of hyper-parameters. To limit this effect, we keep many of the hyper-parameters constant across runs and models. We also set our random seed to ensure random selections across model training instances and to reduce the impact of random initialization. Another threat to validity is that we use the BLEU metric to score and compare our models. The goal of this paper is to show how ensemble models perform compared to current literature. Because BLEU score is so common, this allows us to directly compare our models with most other work in this area.

Beyond the combinatorial explosion of tunable parameters introduced by ensembling models, there are also many ways that models can be ensembled and aggregated. For this paper we chose to focus on stacking and bagging to showcase how basic ensemble methods perform compared to individual baseline models. In this paper we constrain our ensemble models to a maximum of two component models. We do this for a variety of reasons. Primarily, in this paper we aim to explore how complementary and orthogonal models contribute to an ensemble, and as a first step to do this we try to reduce the number of tunable parameters. There are also many ways to aggregate models in an ensemble. We chose a simple, easily explainable aggregation method to help reduce the impact that the aggregation method may have on our results.

V. RQ$_1$: Simple Ensembles

This section describes our methodology for answering RQ$_1$ and our results. Our methodology includes the simple aggregating procedure we used and the baselines we combined.

A. Ensemble Procedure

For our baseline ensemble methods we use two common ensembling procedures from the literature [24]; stacking and bagging. To reduce the number of variables affecting our results, we restrict our ensemble methods to specific sizes and parameters outlined below.

- **Stacking, Figure 2a**: Stacking ensembles aggregate component models that are each trained on the entire training set. Component models in a stacking ensemble do not need to be the same architecture, but they can be. The stacking procedure is 1) select which models will be components in the ensemble, 2) train each model using the entire training set, and 3) select and apply an aggregation method to the output of the models. In our case, each component model in our stacking ensemble is trained on 1.9 million code comment pairs from the Java dataset and a mean aggregation method is applied.
- **Bagging, Figure 2b**: The bagging procedure is very similar to the stacking procedure with one major difference. Instead of training each model on the entire training set, each model is trained on a subset of the training set. There are a variety of ways to choose which subset to train each bagging model on, but a very common method is to randomly select a subset with replacement. In this paper we trained two models on separate randomly selected 50% of the training set. Because the we randomly select with replacement, there may be overlap in the training data for each model. The intuition behind bagging is that a random subset of data may have a different distribution than the dataset as a whole, allowing the model to learn different sets of features.

The aggregation technique we use takes the mean of the output at each time step during inference following work in neural machine translation by Sennrich *et al.* [40]. In their project, they achieved state of the art translation results by ensembling a collection of trained translation models and using the mean of their output vectors to generate a prediction. With this aggregation technique each model will have slight variations on its output distribution, but we can smooth out these variations by averaging the outputs together.

B. Baselines

For our baselines we compare non-ensemble models against stacking and bagging ensemble procedures. We use the baseline models trained following the stacking and boosting procedure as component models in the simple ensemble. We chose these baselines to outline how ensemble methods perform on a variety of model types and architectures, including source code specific features such as the AST and additional file context.

Baseline Models

- **Seq2Seq:** This model is based off the model outlined in Iyer *et al.* [11]. It uses the source code sequence as input into a standard encoder-decoder architecture. We adapt this model by adding an attention mechanism between the encoder and decoder, which has become common practice for sequence-to-sequence models [15].
- **Transformer:** We use a transformer encoder baseline following the current trend in neural machine translation [41]. Transformers have been shown to outperform sequence-to-sequence models that use recurrent layers in translation and summarization tasks. Because they don't generally use recurrent layers they also train faster than similar GRU or LSTM based models. This baseline uses a transformer to encode the source code sequence and a GRU to decode.
- **Seq2Seq-AST-Flat:** This model represents a set of models that use a flattened AST as an input. Hu *et al.* and LeClair *et al.* use an SBT representation of the AST, flattening it to a single sequence. LeClair *et al.* additionally remove identifiers from the SBT representation calling the new representation SBT-AST-ONLY (SBT-AO). We use the SBT-AO representation of the AST for our flat-AST baseline becuase it has been shown to outperform the SBT approach. In this approach both the source code and flattened AST sequence are provided to model as input.
- **Seq2Seq-AST-GNN:** This model uses a GNN to encode the AST instead of flattening it. We follow work outlined in LeClair *et al.*, Fernandes *et al.*, and Xu *et al.* [16], [19], [33]. In these works, the AST is kept as a tree with each AST node becoming an input to the encoder. A GNN layer is then used to encode the AST nodes with attention mechanisms between the source code, AST, and generated comment.
- **Seq2Seq-FC:** The FC version of the seq2seq model follows the work by Haque *et al.* [13] which uses the other methods from the same file as additional context to the model. This additional file context allows the model to learn vocabulary that may not exist within the method itself. Other related work includes additional context information such as API calls [14] or project level information [35]. We chose to use file context because the dataset was readily available and in their work, Haque *et al.* compares to a variety of additional baselines.
- **Seq2Seq-AST-Flat-FC:** Similar to the seq2seq-FC baseline, this baseline uses the file context model proposed by Haque *et al.* [13] with the addition of the flattened AST as an input. This model has three inputs, the source code sequence, the flattened AST sequence, and the additional file context.

C. Results

We present our experimental results in three parts for comparison, 1) baseline models evaluated without ensembling (Figure 3a), 2) combinations of baseline models trained using the stacking procedure (Figure 3b), and 3) combinations of baseline models trained using the bagging procedure (Figure 3c).

In Figure 3a we present BLEU scores for the set of baseline models without any ensembling procedure applied. We observe that the AST-Flat-FC model had the best overall performance with a BA score of 19.31. The Transformer model had the lowest performance with a BA score of 17.74. The Seq2Seq and Seq2Seq-FC models had the same performance with a BA score of 18.15. The AST-Flat model obtained a 19.08 BA score and the AST-GNN model obtained an 18.81 BA score. We use these baseline scores as a direct comparison for the stacking and bagging ensemble procedures. While the results we obtained for the baseline models is in line with those reported in their respective papers, we do see some variation when comparing BA scores. Difference in score can be attributed to random initialization of parameters and differences in training, validation, and testing set split.

In Figure 2a we show the results for the stacking procedure using the set of baseline models as component models. The diagonal of Figure 2a show the results for ensembles whose component models are the same architecture. Outside of the main-diagonal are results for each combination of component models using differing model architectures.

First, if we look only at ensembles whose components are the same architecture (the main diagonal of Fig 2a) we see that the AST-Flat-FC ensemble has the best performance with 20.16 BA score, which is a 4.7% increase over a single AST-Flat-FC model. The ensemble with lowest performance from this group is the Transformer ensemble with 18.88 BA score. Every model, when having components of the same architecture, achieved an improved BA score with an average increase of 1.10 BA, or an average of 6% improvement. The Seq2Seq-FC ensemble had the largest increase in performance with a 1.21 score improvement. We attribute the increase in performance when component models use the same architecture to a smoothing effect that combining the outputs has on the prediction.

Model Type	BA	B1	B2	B3	B4
Seq2Seq	18.15	37.87	20.79	13.58	10.15
Transformer	17.74	36.81	20.14	13.28	10.07
AST-Flat	19.08	38.57	21.78	14.49	10.88
AST-GNN	18.81	37.70	21.33	14.33	10.85
Seq2Seq-FC	18.15	36.87	20.52	13.73	10.44
AST-Flat-FC	**19.31**	**38.62**	**21.83**	**14.70**	**11.22**

(a)

	seq2seq	Transformer	AST-Flat	AST-GNN	seq2seq-FC	AST-Flat-FC
seq2seq	19.34	19.36	19.86	19.81	19.92	20.42
Transformer	x	18.88	19.8	19.64	19.71	20.38
AST-Flat	x	x	20.13	20.13	20.26	**20.64**
AST-GNN	x	x	x	19.89	20.09	20.22
seq2seq-FC	x	x	x	x	19.36	20.20
AST-Flat-FC	x	x	x	x	x	20.16

(b)

	seq2seq	Transformer	AST-Flat	AST-GNN	seq2seq-FC	AST-Flat-FC
seq2seq	18.53	18.43	18.89	18.92	18.97	19.28
Transformer	x	18.07	18.88	18.86	18.82	19.06
AST-Flat	x	x	19.34	18.88	19.34	**19.63**
AST-GNN	x	x	x	19.11	19.30	19.50
seq2seq-FC	x	x	x	x	18.01	18.98
AST-Flat-FC	x	x	x	x	x	18.31

(c)

Fig. 3: BLEU scores for (a) non-ensemble models, (b) stacking with simple aggregation, and (c) bagging with simple aggregation

Second, when we compare ensembles that use component models of different architectures, we see that the AST-Flat-FC+AST-Flat ensemble achieved a BA score of 20.64. This shows an 8.1% improvement over the AST-Flat model and a 6.8% improvement above the AST-Flat-FC model. The worst overall performing ensemble when using two different model architectures is the Seq2Seq+Transformer ensemble. This combination resulted in a BA score of 19.36. The ensemble that had the largest improvement over the baseline model is the Transformer+Seq2Seq-AST-Flat-FC ensemble. This combination achieved a 14.8% BA improvement over the baseline Transformer model, and a 8.9% improvement over the baseline AST-Flat-FC model. The stacking procedure resulted in improvement on every ensemble combination. We believe that the improved score is due to the combination of complimentary and orthogonal information provided by the combination of models.

Figure 3c shows our results from the bagging procedure using the set of baseline models as component models. When the bagging procedure is applied to two model of the same architecture the AST-Flat ensemble has the best BA score of 19.34, which is a 0.03 BA improvement over the baseline AST-Flat model. The Seq2Seq-FC+AST-Flat-FC ensembles see a performance decrease when trained using the bagging procedure. This performance loss can be attributed to the reduced training data size introduced by bagging. The FC models use additional file context inputs to help improve model performance, but this adds additional trainable parameters to the model. We also note that the FC models need to be trained to 15 epochs before convergence as reported by Haque *et al.* while the other models only need 10 epochs to converge. It is likely that these models require more data than the other models due to the number of trainable parameters.

When the bagging procedure is applied to component models of different architectures the AST-Flat+AST-Flat-FC ensemble has the best performance with a BA score of 19.63. The Seq2Seq+Transformer ensemble had the worst overall performance with a BA score of 18.43. We found that the combination of models that had the same input types (e.g. the Seq2Seq and Transformer, or AST-Flat and AST-GNN) generally had minimal performance increase from ensembling. Models that have different or orthogonal data inputs had larger BA score improvements when ensembled.

Overall, we found that bagging performed worse than stacking. This could be due to a variety of factors. First, the complexity of the component models may require a large dataset for model convergence, which bagging reduced the training set size significantly. Second, bagging performance may be sensitive to the number of component models. We limit our work to two component models, but more models may improve performance. The stacking procedure had improvements over all baseline component models with the largest improvement being the Transformer+AST-Flat-FC ensemble, and the best combination of component models being the AST-Flat+AST-Flat-FC ensemble. The stacking ensemble results show that models that were trained using complementary orthogonal input data have the best improvement over their baseline models.

VI. RQ$_2$: Vector Space Analysis

This section describes our methodology for answering RQ$_2$, procedures for analysis, data, and interpretations.

A. Methodology

To answer RQ2 we evaluate and compare how the model encoders create internal representations of the source code. Comparing the output of the model encoders can show us if the models are encoding methods in a similar way. Due to random initialization and weight updates during training, we can not directly compare the output vectors of each model. Instead we compare sets of similar methods using a cosine-similarity score. A high level overview of the process we use to obtain the similar methods can be seen in Figure 4.

Procedure to extract similar methods in the testing set using cosine-similarity:

1) Using a trained model, extract the vector output of the encoder for every method in the testing set. This gives us the models learned internal representation of each method in the testing set. Do this for both encoders in the comparison. This produces two lists of vector representations for each method.
2) For the file context vectors, we average the output of the time distributed GRU layer.
3) For each method vector in the testing set, we find the 100 most similar method vectors using cosine-similarity metric. We apply this to the lists generated by both encoders.
4) To compare the different encoders, calculate the number of functions that overlap between each methods top 100 similarity list.

Using this method we compare the outputs of model encoders for the source code, AST, and file context of the baseline models. We found that the encoders that have many overlapping functions, then it is likely that the encoders have learned to represent methods in a similar way. If there is low agreement between the encoders list of similar functions, then the encoders may have learned orthogonal representations of the source code. Using our results from RQ1 we show that the models that had the largest improvement when ensembled, also show very little overlap in their inputs when compared with other models.

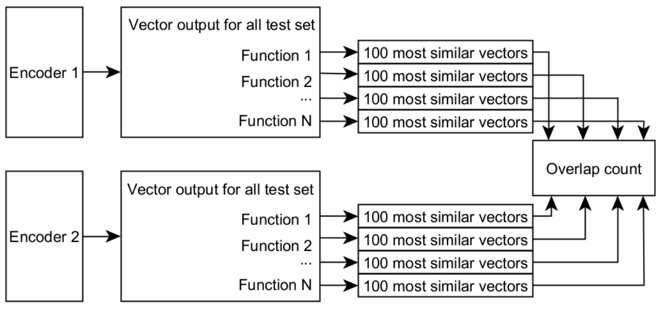


Fig. 4: Methodology for finding orthogonal representations of source code features.

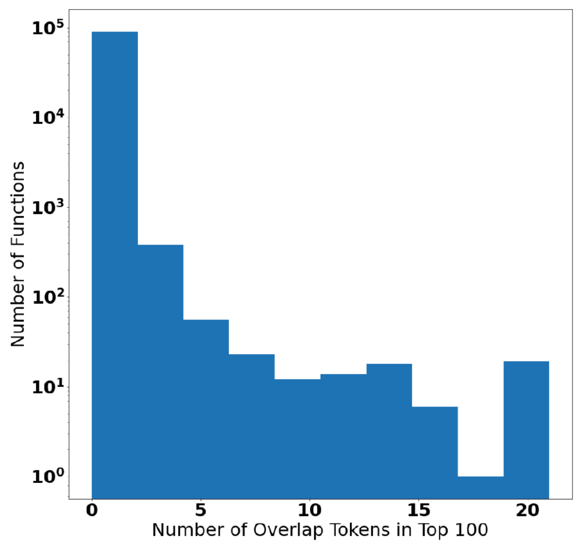
B. Results

Figure 5a shows histograms of encoder comparisons. In these figures the x-axis is the number of methods that overlap in the encoders 100 most similar lists, as explained in the previous section. For example in Figure 5a, the bar labeled '5' is the count of all methods in the testing set that had an overlap count of 5. Having an overlap count of 5 indicates that for a given method the encoders agreed on 5 entries. In this example, the number of methods that had an overlap count of 5 in the testing set is in between 100 and 1000. Each histogram shows the overlap distribution over the testing set.

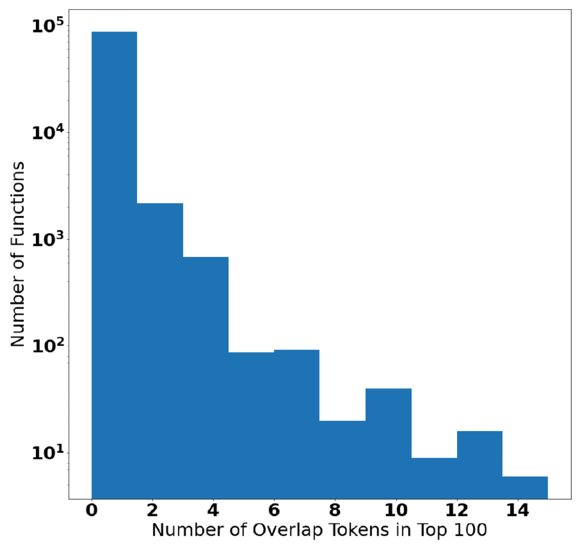
Figure 5a is the overlap between the source code sequence encoder and AST encoder from the AST-Flat model. When comparing the source code sequence encoder to the AST encoder we see very little overlap in method similarity. This may indicate that the source code sequence encoder and the AST encoder have learned to represent methods in different ways. This is likely due to the encoders learning from orthogonal, complementary information. We expect this from inputs that use different parts of the source code, in this case the source code sequence and AST.

In Figure 5b we see a similar situation where there is almost no overlap in the methods the encoders find similar. This figure compares the source code sequence encoder and AST encoder of the AST-GNN model. We see an overlap histogram similar to the AST-Flat comparison, with less overlap in the 10+ groups, but slightly more in the 0-3 groups. This could be due to the GNN providing a more orthogonal representation of the AST than the AST-Flat model. In Figure 5c we show the overlap between the AST encoders of the AST-Flat and AST-GNN models. In this histogram we see that while there is still very little overlap, there are many more methods that have an overlap count of 1 or 2. This could mean that while the ASTs are being represented differently to each model, they are learning some features that allow them to identify a small subset of similar methods the same way.

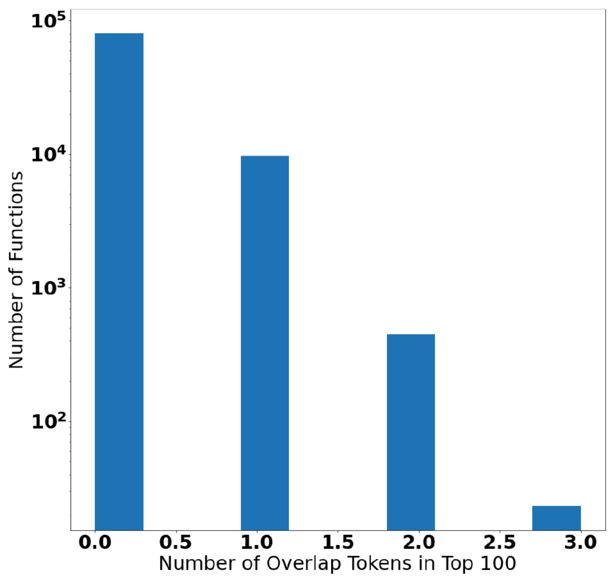
Not all encoder overlaps show orthogonal representations. In Figure 5d there is significantly more overlap between the encoders, with some methods having 70-80 of their top 100 similar methods shared between the encoders. This figure shows the overlap between the source code sequence encoders from the Transformer and AST-Flat models. Both encoders are trained on the source code sequence as input and have learned similar representations of the source code sequence input. We found that many of the of the source code sequence encoders shared a high level of similarity. Similarly, in Figure 5e we compare the file context encoder average output from the Seq2Seq-FC and AST-Flat-FC models. In this comparison we see a lot of overlap between the encoders, the only difference between these two models is that the AST-Flat-FC model has an additional input of the flattened AST. We attribute the large overlap between these encoders to the file context being learned in similar ways between the Seq2Seq-FC and AST-Flat-FC models. It is likely that these models utilize the file context in similar ways when generating summaries.



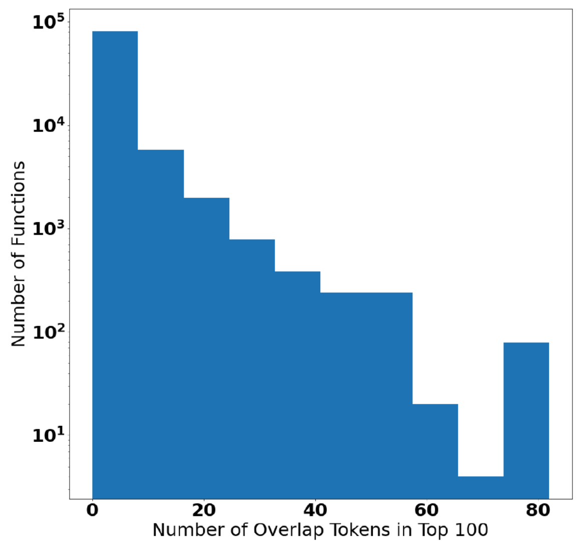
(a) Source code sequence encoder and AST encoder comparison of the AST-Flat model.



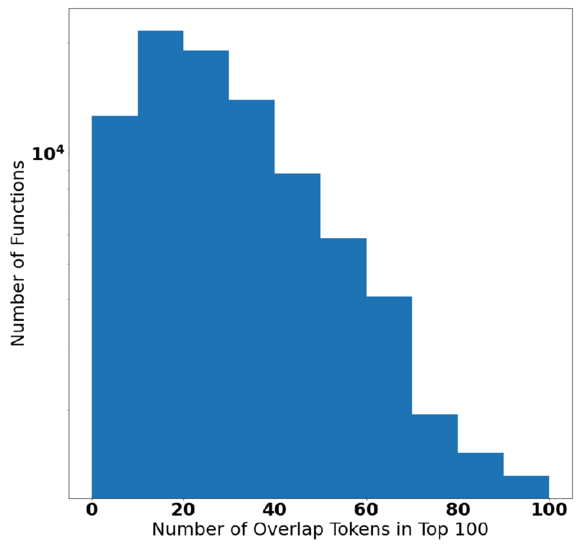
(b) Source code sequence encoder and AST encoder comparison of the AST-GNN model.



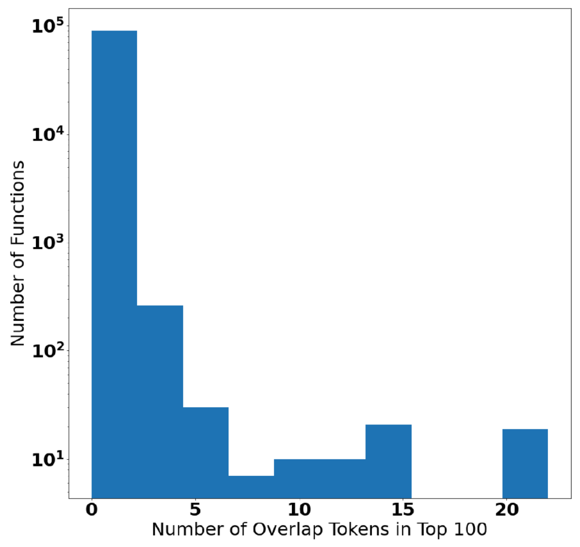
(c) AST encoder comparison of the AST-Flat and AST-GNN models.



(d) Source code sequence encoder comparison of the Transformer and AST-Flat models.



(e) File context encoder from the Seq2Seq-FC and AST-Flat-FC models.



(f) AST encoder for AST-Flat and AST-Flat-FC models.

Fig. 5: The number of methods in the 100 most similar that overlap between encoders

Figure 5f compares the AST encoder of the AST-Flat and AST-Flat-FC models. Again, we see very little overlap between the AST encoders. We found this to be common between all of the models that utilize either flat or GNN representations of the AST. This could be due to each model learning different types of features from the AST that, along with the other inputs, improve model performance. For instance, having the file context along with the AST may allow the model to focus on AST structure elements to boost performance on a specific subset of methods. If the model does not have the file context available, it may have to learn more generalized representations for the AST.

VII. Examples

In this section we give two examples from the testing set. These examples show how the encoders for the source code, AST, and file context differ in how they represent functions and determine which functions are most similar.

A. Example 1

This example compares similar methods from the source code sequence encoder and AST encoder of the AST-Flat model. It shows how the two encoders differ by showing the most similar method to the input method for both encoder. We show the input method source code and source sequence, as well as the associated comment.

The input method is a table GUI method that returns a Boolean if a tooltip is able to be set. The most similar method based on the source code encoder is another GUI method that returns a Boolean and determines if a panel is in a certain position. The most similar method based on the AST encoder is a networking method. This method also returns a Boolean similar to the input method, but it doesn't have much else in common. The AST encoder is finding structural elements to match to. In this case the return type of the method. The source code sequence encoder is matching to similar vocabulary such as 'table', 'graph', 'panel'.

Input Method ID: 18252737

Source code input

```
private boolean showTable(Graph graph,...

  if (table!=null&&!tableNodes.containsKey...

    Node n = graph.addNode();

    n.setString("label", table.getName());

    String tooltip = tableRenderer.get...
    n.setString("tooltip", tooltip);

    tableNodes.put(table, n);
    return true;
  }

  return false;

}
```

Source code sequence

private boolean show table graph graph table table if table null table nodes contains key table node n graph add node n set string label table get name string tooltip table renderer get tool tip table n set string tooltip tooltip table nodes put table n return true return false

Comment

creates visible node for given table

Most similar method based on source code sequence

Similar Method ID: 40467654

Source Code input

public boolean is over panel int ax int ay if tab only return is over ax ay else point p new point 0 0 calc abs position p if ax p x ax p x width ay p y tab height ay p y height return true else return false

Comment

determines whether the position ax ay is over the panel takimg

---

Most similar method based on AST sequence

Similar Method ID: 39298423

Source Code input

private boolean check target reconnection line endpoints must be different shapes if new target equals old source return false return false if the line exists already for iterator iter new target get target segment part delegates iterator iter has next segment part delegate conn segment part delegate iter next return false if a old source new target line exists already and it is a differenct instance that the line field if conn get point adelegate equals old source conn equals line return false return true

Comment

return true if reconnecting the line instance to new source is allowed

B. Example 2

In this example we compare the closest methods from the source code sequence encoder and the file context encoder using the AST-Flat-FC model. This model uses a source code sequence, AST, and file context as input. The input method is a GUI method that creates a panel for cvs options. When using the source code sequence encoder to find the most similar method, we get a method that sets a view for a model. This likely was the closest method for this encoder because of tokens such as 'position', 'window', 'grid' which all are used in many GUI methods. The file context encoder also finds a GUI method that shares language with the input method such as 'minimizer' and 'layout'. The file context provides additional vocabulary context, unlike the AST which learns structural similarities.

Method ID: 299963

Source code input

```
private void createCVSOptions(int timeWindow){
  this.timeWindow = new TextField(
                 Integer.toString(
                      timeWindow));
  CVSOptions = new Panel();
  CVSOptions.setLayout(new GridBagLayout());
  GridBagConstraints c = new GridBagConstraints();
  c.anchor = GridBagConstraints.WEST;
  c.fill = GridBagConstraints.NONE;
  c.weightx = 1;
  c.weighty = 1;
  c.gridx = 0;
  c.gridy = 0;
  CVSOptions.add(new Label("Time window:"), c);
  c.gridx = 1;
  CVSOptions.add(this.timeWindow,c);
}
```

Source code sequence

private void create cvsoptions int time window this time window new text field integer to string time window cvsoptions new panel cvsoptions set layout new grid bag layout grid bag constraints c new grid bag constraints c anchor grid bag constraints west c fill grid bag constraints none c weightx 1 c weighty 1 c gridx 0 c gridy 0 cvsoptions add new label time window c c gridx 1 cvsoptions add this time window c

Comment

construct the panel for the cvs options

Most similar method from source code sequence encoder

Method ID: 45891192

Source Code input

public void show big view state model model state view small small view m current big view new state view big model small view this m current small view small view todo move controller like state controller small is instantiate new state controller big m model this model m current big view m right panel remove all m right panel add m current big view border layout center m model set current position index of view small view refresh

Comment

sets the model for the detailed view

Most similar method from file context encoder

Method ID: 299982

Source Code input

private void enable minimizer options boolean b dim set enabled b iter set enabled b init layout set enabled b attr exp set enabled b repu exp set enabled b grav set enabled b no weight set enabled b vert repu set enabled b load init layout set enabled b

Comment

enable the part concerning the minimizer

VIII. Discussion & Future Work

In this paper we present two major additions to the work in source code summarization.

1) We explore the performance of ensembling a variety of baseline models using a simple aggregation technique to show how models combined with different architectures and inputs perform on the task of source code summarization.
2) To help explain why ensembling may work well with models that use orthogonal types of input data, we explore and compare the internal source code representations these models learned. This provides insight into how we may be able to better combine models, as well as which types of models may perform best when ensembled.

In our encoder representation comparison we discuss why complementary orthogonal input may be beneficial for models to learn better source code representations. We also show two examples to further illustrate how the learned representations of source code differs between encoders that were trained on different features of source code data (source text, AST, file context, etc). Through these examples we can see that models trained on orthogonal data complement each other well when ensembled. Also, even when training models of the same architecture, ensemble methods improve overall model performance. This paper provides a groundwork for future projects focused on the problem of ensembling source code summarization models.

A. Future Work

One potential path for future work is to explore aggregation strategies and how they can be optimized for source code summarization. In this paper we use a simple mean combination aggregation strategy and do not explore how more advanced aggregation strategies, such as meta-learning, may perform.

IX. Reproducibility

All of our models, source code, and data used in this work can be found in our online repository at https://bit.ly/3tiF8pc

A. Hardware Details

For training, validating, testing of our models we used a workstation with Xeon E1430v4 CPUs, 110GB RAM, a Titan RTX GPU, and a Quadro P5000 GPU

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] 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.

[2] 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.

[3] L. Shi, H. Zhong, T. Xie, and M. Li, "An empirical study on evolution of api documentation," in *International Conference on Fundamental Approaches To Software Engineering*. Springer, 2011, pp. 416–431.

[4] H. Zhong and Z. Su, "Detecting api documentation errors," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 803–816.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] X. Song, H. Sun, X. Wang, and J. Yan, "A survey of automatic generation of source code comments: Algorithms and techniques," *IEEE Access*, 2019.

[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] 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)*, 2016, pp. 2073–2083.

[12] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "A neural architecture for generating natural language descriptions from source code changes," *arXiv preprint arXiv:1704.04856*, 2017.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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

[18] 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

[19] K. Xu, L. Wu, Z. Wang, Y. Feng, M. Witbrock, and V. Sheinin, "Graph2seq: Graph to sequence learning with attention-based neural networks," *Conference on Empirical Methods in Natural Language Processing*, 2018.

[20] 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.

[21] R. Dabre, C. Chu, and A. Kunchukuttan, "A survey of multilingual neural machine translation," *ACM Computing Surveys (CSUR)*, vol. 53, no. 5, pp. 1–38, 2020.

[22] M. Z. Hossain, F. Sohel, M. F. Shiratuddin, and H. Laga, "A comprehensive survey of deep learning for image captioning," *ACM Computing Surveys (CsUR)*, vol. 51, no. 6, pp. 1–36, 2019.

[23] S. Wang and X. Yao, "Diversity analysis on imbalanced data sets by using ensemble models," in *2009 IEEE symposium on computational intelligence and data mining*. IEEE, 2009, pp. 324–331.

[24] O. Sagi and L. Rokach, "Ensemble learning: A survey," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 4, p. e1249, 2018.

[25] S.-L. Hsieh, S.-H. Hsieh, P.-H. Cheng, C.-H. Chen, K.-P. Hsu, I.-S. Lee, Z. Wang, and F. Lai, "Design ensemble machine learning model for breast cancer diagnosis," *Journal of medical systems*, vol. 36, no. 5, pp. 2841–2847, 2012.

[26] S. Rasp and S. Lerch, "Neural networks for postprocessing ensemble weather forecasts," *Monthly Weather Review*, vol. 146, no. 11, pp. 3885–3900, 2018.

[27] S. Borovkova and I. Tsiamas, "An ensemble of lstm neural networks for high-frequency stock market classification," *Journal of Forecasting*, vol. 38, no. 6, pp. 600–619, 2019.

[28] S. Al-Dahidi, O. Ayadi, M. Alrbai, and J. Adeeb, "Ensemble approach of optimized artificial neural networks for solar photovoltaic power prediction," *IEEE Access*, vol. 7, pp. 81 741–81 758, 2019.

[29] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "A neural architecture for generating natural language descriptions from source code changes," in *ACL*, 2017.

[30] Y. Lu, Z. Zhao, G. Li, and Z. Jin, "Learning to generate comments for api-based code snippets," in *Software Engineering and Methodology for Emerging Domains*. Springer, 2017, pp. 3–14.

[31] Y. Liang and K. Q. Zhu, "Automatic generation of text descriptive comments for code blocks," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

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

[33] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," *CoRR*, vol. abs/1811.01824, 2018. [Online]. Available: http://arxiv.org/abs/1811.01824

[34] 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.

[35] A. Bansal, S. Haque, and M. C., "Project-level encoding for neural source code summarization of subroutines," in *29th IEEE/ACM International Conference on Program Comprehension (ICPC'21)*, 2021.

[36] E. Garmash and C. Monz, "Ensemble learning for multi-source neural machine translation," in *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*. Osaka, Japan: The COLING 2016 Organizing Committee, Dec. 2016, pp. 1409–1418. [Online]. Available: https://www.aclweb.org/anthology/C16-1133

[37] M. Asad, "Optimized stock market prediction using ensemble learning," in *2015 9th International Conference on Application of Information and Communication Technologies (AICT)*, 2015, pp. 263–268.

[38] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, 1997. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S002200009791504X

[39] 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.

[40] R. Sennrich, O. Firat, K. Cho, A. Birch, B. Haddow, J. Hitschler, M. Junczys-Dowmunt, S. Läubli, A. V. Miceli Barone, J. Mokry, and M. Nadejde, "Nematus: a toolkit for neural machine translation," in *Proceedings of the Software Demonstrations of the 15th Conference of the European Chapter of the Association for Computational Linguistics*. Valencia, Spain: Association for Computational Linguistics, April 2017, pp. 65–68. [Online]. Available: http://aclweb.org/anthology/E17-3017

[41] 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.