|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SI | Paper | Problem | Insights - Inspiration for the solution | Steps/ input output |
| 1 | [Code Structure–Guided Transformer](https://dl.acm.org/doi/full/10.1145/3522674) | LLM  Code summarization  AST's fed into RNN may not capture long range dependencies between code tokens.  GNN could be one of the solutions, but they rely on predefined Graphs, causing scalability issues | integrating both the detailed local structure (such as individual code tokens and statements) and the overall global structure (like data flow between different parts of the code) could improve the performance of AI modelsfor code summarization. They noticed that current models either miss these relationships (RNN) or handle them poorly (GNN), which limits their effectiveness. | Creates a Structure Guided Transformer SG Transformer Model  Input: The input to their model includes the code itself, along with its detailed structure (like individual tokens and statements) and the overall structure (like how data flows through the code).  Output: The output is an improved, more accurate code summary that better reflects both the small details and the big picture of the code's functionality. |
| 2 | [Towards Summarizing Code Snippets Using Pre-Trained Transformers](https://arxiv.org/abs/2402.00519) | LLM Approach | Problem: Most existing tools summarize whole functions or methods, but sometimes developers need summaries for small bits of code (like just a few lines or snippets). The tools out there don’t do this very well.  Inspiration for the Solution: The authors realized that linking small comments to specific code snippets is hard, but if they could create a dataset of snippet-comment pairs and train a model on that, they could help developers summarize code snippets more effectively. | Steps/Input-Output:  Input: A dataset of code snippets and their matching comments. Steps: Build a dataset of code snippets and comments that explain them. Train a model to understand and generate summaries for snippets based on this dataset. Output: The model, called STUNT, is better at summarizing small code snippets than previous models. It can now automatically generate useful descriptions for individual code lines. |
| 3 | Naturalness of Attention: Revisiting Attention in Code Language Models | LLM | When we use attention mechanisms in models like CodeBERT to understand code, we usually only look at the attention weights, but this doesn't give us the full picture of how the model really "understands" code.  The authors realized that there’s more going on than just attention weights—like how the model transforms inputs. By looking at these transformations, we can get a better idea of how the model captures the structure of code, not just focus on the attention itself. | Input: CodeBERT models trained on Java and Python code. Steps: Analyze how both attention weights and transformations work in the model. Compare how these factors capture the structure of code. Output: Turns out, paying attention (pun intended) to how the input is transformed helps us understand the model better, especially in terms of how it captures code structure. |
| 4 | Enhancing Code Understanding for Impact Analysis by Combining Transformers and Program Dependence Graphs | LLM | Problem: In large software systems, even a small code change can lead to unintended side effects. Figuring out which parts of the system will be affected by such changes, known as impact analysis (IA), is hard to do manually. Most automatic tools are either too slow or not accurate enough because they rely on limited information like static analysis or commit histories.  Inspiration for the Solution: The authors realized that combining semantic information (what the code does) with structural information (how different parts of the code are connected) could lead to a more accurate impact analysis. They drew inspiration from recent advances in transformer models for code understanding, like CodeBERT and GraphCodeBERT, and decided to combine these models with program dependence graphs (which show how code entities are related). | Steps/Input-Output:  Input: Java source code files, with methods and dependencies between them. Steps: Build a program dependence graph that shows which methods depend on each other. Use transformer models like GraphCodeBERT to extract semantic embeddings (representations of what the code does) for each method. Combine the method’s semantic information with the information from the program dependence graph using an embedding propagation strategy. This updates the representation of each method by including info from its related methods. Compare these updated embeddings using cosine similarity to create a ranked list of methods that are most likely impacted by a change. Output: The tool, Athena, provides a ranked list of methods that could be affected by a code change, helping developers focus on the parts of the code that matter the most. The method improves the accuracy of IA by 10-11% over previous approaches. |
| 5 | [Automating Code-Related Tasks Through Transformers: The Impact of Pre-Training](https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10172872) | LLM | Problem: Pre-trained transformers, like those used in natural language processing (NLP), are increasingly popular for automating code-related tasks. However, there’s not enough understanding of how different pre-training objectives (like masking parts of the code) impact their performance for tasks such as bug fixing, code summarization, and code completion.  Inspiration (Insights): The authors wondered whether specialized pre-training objectives (designed for code-specific tasks) could improve the model's performance more than the standard approaches, like the Masked Language Model (MLM) commonly used in transformers. They hypothesized that if the pre-training is more closely aligned with the task at hand, the model might perform better. | Steps/Input-Output:  Input: Transformers pre-trained using both generic and code-specific objectives, then fine-tuned on tasks like bug-fixing, code summarization, and code completion. Steps: Conduct a systematic review to identify commonly used pre-training objectives in software engineering tasks. Pre-train 32 transformers using different objectives, both generic (like MLM) and specific to tasks like method name generation for code summarization. Compare the performance of these models with non-pre-trained transformers. Output: They found that pre-training helps when there’s not much fine-tuning data, but if you have plenty of fine-tuning data, it doesn’t add much. Also, generic pre-training objectives like MLM usually work just fine. |