Which Syntactic Capabilities Are Statistically Learned by Masked Language Models for Code?

Alejandro Velasco, David N. Palacio, Daniel Rodriguez-Cardenas and Denys Poshyvanyk
{svelascodimate,danaderpalacio,dhrodriguezcar,dposhyvanyk}@wm.edu
William & Mary
Williamsburg, Virginia, USA

ABSTRACT

This paper discusses the limitations of evaluating Masked Language Models (MLMs) in code completion tasks. We highlight that relying on accuracy-based measurements may lead to an overestimation of models' capabilities by neglecting the syntax rules of programming languages. To address these issues, we introduce a technique called Syntax*Eval* in which *Syntactic Capabilities* are used to enhance the evaluation of MLMs. Syntax*Eval* automates the process of masking elements in the model input based on their Abstract Syntax Trees (ASTs). We conducted a case study on two popular MLMs using data from GitHub repositories. Our results showed negative causal effects between the node types and MLMs' accuracy. We conclude that MLMs under study fail to predict some syntactic capabilities.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

deep learning, code generation, interpretability, transformers, dl4se

ACM Reference Format:
Alejandro Velasco, David N. Palacio, Daniel Rodriguez-Cardenas and Denys Poshyvanyk. 2024. Which Syntactic Capabilities Are Statistically Learned by Masked Language Models for Code? . In *New Ideas and Emerging Results (ICSE-NIER'24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3639476.3639768

1 INTRODUCTION

Large language models have illustrated convincing performance across a range of different software engineering (SE) tasks [5, 7, 23, 35, 36, 39–41]. In particular, *code generation* has been an important area of research for SE tasks such as code completion [8]. *Code completion* is a disciplined technique for generating missing *syntactic features* of an incomplete snippet based on its semantic and structural context [4]. These syntactic features usually adopt the form of identifiers, function names, conditionals, or parameters depending on the granularity of the snippet. Software researchers are particularly interested in improving code completion to optimize time spent during the development and maintenance cycles [12, 13]. Numerous studies have investigated code completion automation using machine learning [4, 15, 28, 42]. Current research has focused on exploiting deep learning representations using LSTMs [33], GPT [32], RoBERTa [20], and T5 [6, 9].

Masked Language Models (MLMs) have been recently used for code completion tasks demonstrating promising results (an avg. accuracy of 38.7% in perfect predictions) at different masking levels (*i.e.,* Token, Construct, and Block) [6]. Some studies suggest that MLMs statistically learn the underlying structure of Abstract Syntax Trees (ASTs) at certain degree [19, 24, 37]. Yet, given the high accuracy achieved by MLMs [6], few attempts have been made to investigate the role of **Syntactic Capabilities** for evaluating code completion. Syntactic Capabilities are *interpretable* prediction estimates for a terminal (N) and non-terminal (Σ) nodes of ASTs that are ruled by a *Context Free Grammar* (CFG) of Programming Languages (PLs) [31].

To date, the primary focus on evaluating MLMs has been on the role of *accuracy* as the principal metric, which may lead to erroneous and/or incomplete interpretation of the *syntactic features* embedded in neural architectures [25, 27, 37]. Relatively little is understood about incorporating these interpretable prediction estimates into the evaluation of MLMs, hence *current evaluation methods do not help practitioners to decide* whether MLMs are confidently generating code at AST node granularity and to what extent these syntactic features affect general prediction performance. That is, these methods do not reveal information about syntactic capabilities and their causal effects on the overall MLMs performance.

Our study attempts to establish the causal connection between syntactic features in the form of AST node types and MLMs' performance. Under this premise, we introduce Syntax*Eval*, an approach that leverages syntactic capabilities to evaluate how good MLMs infer N and Σ AST nodes of a given PL. When evaluating the performance of an MLM, Syntax*Eval* selectively masks tokens according to the AST Node types defined by the CFG. Subsequently, an MLM predicts the masked tokens. Finally, Syntax*Eval* measures the **causal effect** of AST node types on code completion performance.

Our results suggest that although MLMs are homogeneously predicting individual AST node types with high accuracy, we observed no evidence of effects from syntactic features on MLMs' prediction after controlling for confounding factors. Hence, no causal evidence supports the fact that MLMs are statistically learning syntactic structures with acceptable confidence, contradicting recent studies in the explainability field [24, 37]. We hope that the results of our work will shed more light on the syntactic capabilities of current MLMs to enable a more systematic and rigorous evaluation of code completion tasks. The contributions of this paper are as follows: 1) a technique for evaluating the extent to which MLMs

ICSE-NIER'24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0500-7/24/04.
https://doi.org/10.1145/3639476.3639768

predict AST structures; 2) a case study that leverages causal analysis to understand how different AST node types influence code completion; 3) experimental data, curated datasets, source code, and complementary statistical analysis used in this research are published in an open-source repository [1].

2 BACKGROUND & RELATED WORK

The accurate identification and generation of code tokens is a widely studied field at the intersection of SE and DL [38]. State-of-the-art code generators estimate the token prediction using probabilistic distribution (*i.e.,* a Large Language Model (LLMs)) obtained by training on large amounts of code corpora. Put simply, code completion models should statistically approximate the production rules defined by the CFG. These production rules are recursively applied to terminal N and non-terminal Σ nodes to formally define the structure of a PL. For instance, recent explainability studies have claimed that the syntactic structures of code are encapsulated in the internal layers of LLMs across software tasks, implying a foundational statistical comprehension of code semantics [14, 37]. In this section, we introduce the concept of MLMs and their current evaluation methods.

Masked Language Models for Code. Considerable research attention has been directed toward the usage of Bidirectional Encoder Representation from Transformers (BERT) on code completion as an attempt to push the predictability boundaries beyond the next token prediction. BERT allows higher granularity syntax structures (*i.e.,* entire code statement) to be generated using self-attention layers trained to restore a masked subset of tokens in the input [6, 10]. This peculiar form of training the architecture is known as *denoising autoencoding*, or Masked Language Models (MLM), which we formalize as $MLM(C) = \mathbb{E}_{s\in S}\mathbb{E}_{M\subset s}\left[\sum_{s_j\in M}\log p(s_j|\tilde{s})\right], |M| = |m(s)|$, where a masking rate m (usually 15%) is applied on the original sequence s of a training corpus S. The model attempts to predict the set of masked tokens M given the corrupted context $\tilde{s}$ (the masked version of s) [18]. MLMs for code completion are mostly evaluated using metrics such as CodeBleu, EM, F1, and Pass@k [16].

Syntax-Based Evaluation of MLMs. Due to the unpredictable behavior of MLMs while generating tokens, explainability techniques are complementary evaluative methods for understanding the decision-making process by reducing the uncertainty of the models. Such uncertainty can be controlled by exploring the inner layers of the neural net or performing guided perturbations on models' input [3]. Recent studies have explored the use of structural information as an interpretability tool for pre-trained models for code [25]. For instance, Wan et al. [37] conducted an explainability analysis focusing on three aspects: 1) how the self-attention weights align with the syntax structure, 2) whether the syntax structure is encoded in the hidden layers, and 3) how pre-trained models induce syntax structures. Similarly, Mohammadkhani et al. [24] propose an eXplainable AI method (attention mechanism) on three downstream code tasks: 1) code generation, 2) refinement, and 3) translation. Previous findings imply that Encoder-based models can effectively extract detailed syntactic information using self-attention mechanisms. We used prior observations about encoded information of ASTs to formulate an evaluative approach based on measuring the prediction performance of syntactic capabilities directly from (non)terminal nodes.

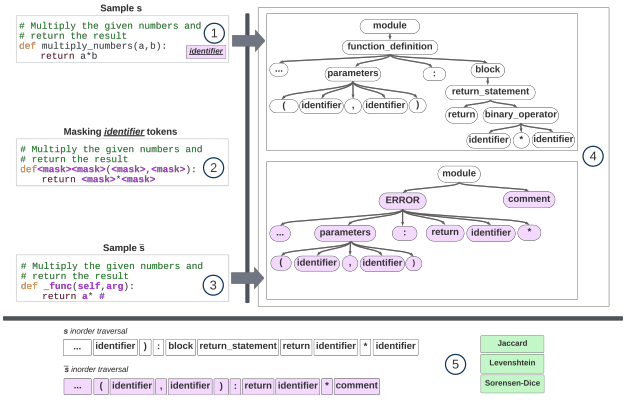

Figure 1: Syntax*Eval* Process for the *identifier* AST Node.

3 SYNTACTIC CAUSAL EVALUATION

Syntax*Eval* is an evaluative approach organized into two distinct parts. The first part estimates a fine-grained performance, grouped by AST node types, for a given MLM (**RQ**$_1$). The second part adopts causal interpretability theory to quantify the influence of previously estimated AST node types on the accuracy of the model (**RQ**$_2$).

Evaluating Syntactic Capabilities. Fig. 1 depicts the process of evaluating syntactic capabilities for code completion using MLMs. This evaluative process is comprised of five steps. Firstly, we must define a set of AST node types C to be analyzed. This set of AST node types is ruled by Python CFG adopting the form of $C = N \cup \Sigma$. Then we search for the positions of these node types C in the code sequence after iterating for each sample s_i (*i.e.,* snippet) of a given ground truth S (Fig. 1- ①). Secondly, detected tokens, which correspond to the previously defined set C, are masked with the label <**mask**> (Fig. 1- ②). Thirdly, we use an MLM to infer the masked tokens for each sample s_i obtaining a set $\bar{S}$ of predicted samples $\bar{s_i}$. Fourthly, we parse the AST of s_i and $\bar{s_i}$ to generate a list of extracted nodes for the ground truth and predicted samples using the in-order traversal algorithm (Fig. 1- ④). Finally, we compare both ground truth s_i and predicted $\bar{s_i}$ lists of extracted nodes by computing three similarity metrics for each sample (*i.e.,* **Jaccard**, **Levenshtein** & **Sorensen-Dice**) (Fig. 1- ⑤).

Computing Causal Interpretability. Causal Inference has been adopted to complement the assessment of LLMs by controlling for confounding factors in code data. Palacio et al. [25] introduce do_{code}, a post hoc interpretability methodology that explains model predictions by providing causal explanations. These explanations are generated by estimating the effect of binary interventions T, such as masking random tokens T_0 versus masking AST node types T_1, on MLMs' performance. Specifically, in Syntax*Eval*, the treatment T_1 refers to samples that are masked on AST node tokens C, while the control T_0 refers to samples that are randomly masked on any position. The control T_0 preserves the same number of masked tokens as in T_1.

Syntax*Eval* formulates a **Structural Causal Model** (SCM), which is a graphical model composed of outcomes, treatments, and confounders [26], to explain a set of *potential outcomes* Y (*e.g.,* Jaccard, Levenshtein, Sorensen-Dice) in terms of treatments T (*i.e.,*

Table 1: Evaluated Encoder-Based Transformers.

Id	MLM	Size	Layers	Vocab.
M_1	CodeBERTa-small-v1 [21]	84M	6	52,000
M_2	codebert-base-mlm [11]	125M	12	50,265

masked AST node types) by controlling for a set of code confounders Z to avoid *spurious correlations*. These code confounders consist of seven variables, which include the # of parsing errors, the height of the AST, the # of nodes, the # of whitespaces, the # of lines of codes, the cyclo complexity, and the token counts. Finally, Syntax*Eval* computes the *Average Treatment Effect* (τ) of a treatment T has on the outcomes Y after controlling for confounders Z. In other words, we want to estimate the expected value $\tau = \mathbf{E}[\tau(Z)] = \mathbf{E}[Y|do(T_1)] - \mathbf{E}[Y|do(T_0)] = \mathbf{E}[Y_1 - Y_0]$. The variables Y_1, Y_0 refer to potential outcomes observed under the treatments T_1, T_0. For the sake of brevity, we do not discuss the details of treatment effects computations. However, these effects are approximated using *propensity score methods* after applying the *the back-door criterion* [30].

4 CASE STUDY DESIGN

This section outlines the methodology employed to consider the potential influence of syntactic capabilities on the evaluation of MLMs, we conducted a case study on two popular architectures to explore the following RQs:

RQ$_1$ [Performance] *How good are MLMs at predicting AST nodes?*

RQ$_2$ [Causality] *How do node types impact MLMs' performance?*

Data Collection: To mitigate the risk of data snooping, we curated our testbed with $50k$ Python snippets. This testbed exclusively comprises commits executed between January 01, 2022 and January 01, 2023. We collected the snippets from newly added or updated Python Github repositories with over 1k stars scoring. Additionally, we discarded duplicated samples by referring to the history of the commits. The testbed also contains complementary code features (*e.g.,* LoC, CYCLO, and # of nodes), these features were extracted using *Galeras* pipeline [29]. **Masked Language Models:** We evaluated two encoder-based transformers trained on *CodeSearchNet* [17] with different hyperparameters (see Tab. 1). These encoders have been assessed in prior studies in which they were found to capture structural information [37], [37], and [24]. **Node Types:** Tree-sitter CFG defines 196 AST node types C for Python. For the sake of simplicity, we selected a subset of terminal and non-terminal nodes defined in Python's CFG as depicted in the first column of Tab. 2. The subset entails the most basic syntactic structures for control, iteration, operators, and functional programming. This study showcases the nodes that exhibited the most interesting behavior. We chose Python for code completion experiments due to its extended use in recent studies.

Evaluation Methodology. To address **RQ$_1$**, we estimated syntactic capabilities of M_1 and M_2 encoders using $8K$ randomly selected samples from the collected testbed. Syntax*Eval* masks the associated tokens for each chosen node type (T_1) and subsequently uses the MLM to infer the missing elements. Then, we compute normalized similarity distances (*i.e.,* Jaccard, Levenshtain, and Sorence-Dice) between the AST in-order traversal of both the predictions and the ground truth. Global results indicate the average prediction accuracy (*i.e.,* normalized distance) for all node types within C. In contrast, local results detail the prediction accuracy for individual node types.

To address **RQ$_2$**, Syntax*Eval* computes the *Average Causal Effect* between syntactic capabilities and MLMs' performance. This method consists of estimating τ using treatments T_1 and T_0 (*i.e.,* tokens randomly masked) while controlling for confounders in Z (code features in Data Collection), to mitigate the presence of *spurious correlations*. The removal of confounding bias can be formally achieved using both an SCM and the *do*-operator introduced by Pearl et al. [26]. To verify the robustness of our SCM, we computed *placebo* refutations, which is a method that fakes an unrelated treatment by re-estimating the causal effects. That is, we assessed that the causal effects of the fake treatment on the outcome were close to zero. Moreover, to ensure a balanced distribution of randomly masking tokens within T_0, we created 20 distinct variations for each sample. Afterward, we computed the average of the resulting similarity scores. Finally, to ensure statistical significance, we *bootstrapped* the similarity scores using the *mean* for 500 samples per node type.

5 RESULTS & DISCUSSION

The aim of this study is to determine the effect of Syntactic Capabilities, in the form of interpretable prediction estimates for node types, on the prediction performance of MLMs. We concentrated on evaluating Encoder-based Transformers beyond accuracy.

5.1 RQ$_1$ Syntactic Capabilities Performance

Global Results. A cursory glance at Tab. 2 reveals that control groups T_0 of each performance metric are *not significantly different* from treatments T_1 for both encoders. For example, the control median values greater than 0.8 are within the interquartile range (*iqr*) 0.78 ± 0.22 of the corresponding treatment. Furthermore, the standard deviation (*std*) values of the performance are predominantly more dispersed in the treatments than in the control. For example, the T_1 *std* of M_1 Jaccard is 0.21, while the T_0 is 0.17. Appealingly, all average values of performance are above 0.5, this indicates that M_1 and M_2 models are predicting masking tokens with high confidence despite the group treatments T. Although the median global performance has consistently high accuracy among the metrics (> 0.8), the average separation values between the T groups are not significant with an average median distance of 0.096 and 0.06 for M_1 and M_2 respectively. However, a preliminary analysis for node types estimations suggests that M_1 and M_2 have a tendency to not statistically learn syntactic-oriented masked tokens T_1. Our findings reveal a subtle inclination towards predicting random masked tokens over syntactic-oriented ones.

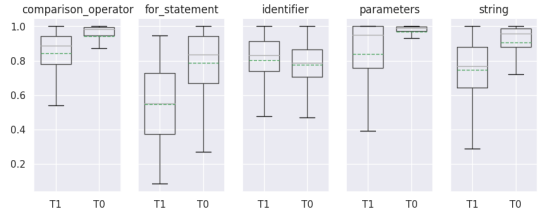

Figure 2: T_0 vs. T_1 Local Jaccard for *Nodes* using M_1.

Table 2: Global Perf. and Causal Effects for M_1 and M_2.

Performance	Jaccard		*Levenshtein*		*Sorensen-Dice*	
MLMs	M_1	M_2	M_1	M_2	M_1	M_2
Treatments	***Performance Metric Y [avg ± std]****					
T_0	0.88 ± 0.17	0.84 ± 0.16	0.87 ± 0.16	0.83 ± 0.17	0.92 ± 0.1	0.89 ± 0.12
T_1	0.78 ± 0.21	0.76 ± 0.21	0.78 ± 0.22	0.76 ± 0.21	0.85 ± 0.17	0.84 ± 0.17
AST Node Type C	***CausalEffect τ***					
boolean_operator	-0.083	-0.150	-0.069	-0.136	-0.048	-0.095
comparison_operator	-0.186	-0.027	-0.179	-0.018	-0.126	-0.015
for_in_clause	-0.059	-0.053	-0.050	-0.045	-0.034	-0.029
for_statement	-0.269	-0.101	-0.193	-0.041	-0.243	-0.083
identifier	0.016	-0.075	0.001	-0.073	0.010	-0.039
if_clause	-0.070	-0.040	-0.058	-0.036	-0.037	-0.022
if_statement	-0.163	-0.118	-0.140	-0.093	-0.116	-0.095
parameters	-0.140	-0.048	-0.127	-0.046	-0.087	-0.029
return_statement	-0.144	-0.121	-0.118	-0.113	-0.087	-0.075
string	-0.156	-0.168	-0.102	-0.145	-0.118	-0.116
while_statement	-0.200	-0.096	-0.139	-0.009	-0.186	-0.077

* Medians are > 0.8. The biggest causal effect τ for each node type is in gray.

Local Results. Fig. 2 shows the Jaccard performance statistical behavior across some selected node types for M_1. Due to the non-overlapping *iqr* between the T_0 and T_1, we observed a significant difference between treatment groups in the performance distribution for the nodes *comparison_operator* and *string*, revealing that M_1 struggles at predicting tokens associated with such types in contrast to random masked tokens. We found that *identifier* was the only node type that performed better in the treatment than the control group. Fig. 3 presents the Empirical Cumulative Distribution (ECD) plots of T_1 Jaccard distance across selected node types. We observed that *if_clause* was remarkably achieved with the highest score prediction (0.9) at the lowest percentage of the population (42% of the samples in the testbed). Conversely, *for_statement* was the most difficult node to predict across the population. We believe that MLMs struggle to predict these previous nodes due to their complexity. A node is complex when its block has incorporated other node types.

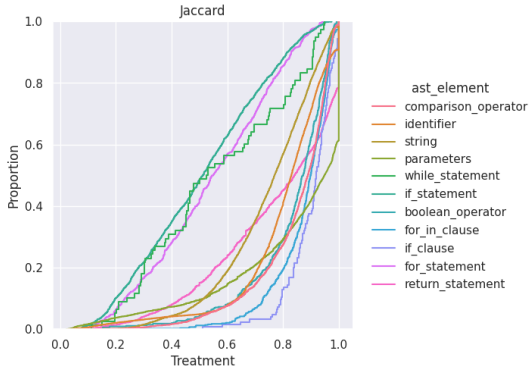

Figure 3: Syntactic Capabilities Statistically Learned by M_1.

> ***RQ***$_1$*:* MLMs tend to complete missing AST-masked tokens with acceptable accuracy (> 0.5). However, the reported performance suffers from high variability (±0.21) making the prediction process less confident compared to completing randomly masking tokens.

5.2 RQ$_2$ Causal Evaluation Effect

This study used a quantitative causal technique to analyze the influence of masking binary treatments (*i.e.,* AST and random) on the performance of both M_1 and $M2$ transformers after defining the Structural Causal Model of the problem. To draw a causal link between syntactic features (*i.e.,* AST nodes) and performance metrics (*i.e.,* Jaccard, Levenshtain, and SD), we expect to observe a **positive causal effect**. A positive effect would indicate that syntactic features C are affecting models' performance and AST nodes would be statistically learned by MLMs. On the other hand, a **negative causal effect** would imply that randomly masked tokens have more influence on the performance. That is, tokens without any particular syntactic order are being predicted accurately.

Unlike previous assumptions, it can be inferred from Tab. 2 that the control group (*i.e.,* masking random treatment) is having more impact on MLMs' performance than the actual syntactic features. For instance, a set of samples masked for *for_statement* tokens are underperforming (*a.k.a.* negative effects) compared to the same set but randomly masked tokens. This suggests that although transformers are predicting AST node types with confidence (see Fig. 3, these syntactic features are not particularly relevant compared to predicting any other set of unstructured tokens in the snippet (see gray areas in Tab. 2). These findings tend to corroborate Karmakar et al. research [19] in which MLMs do not fully grasp the syntax and structural aspects of code. Our findings offer an alternative perspective compared to claims made by other probing approaches [22, 34]. For example, Hernandez Lopez et al. [14] argue for the presence of a syntax subspace within the hidden layers that encode structures of PLs. Similarly, Toufique et al. [2] outline that pre-trained language models learn robust representations of code semantics, which implies a deep understanding of syntax elements from the source code.

> ***RQ***$_2$*:* The performance of MLMs is negatively impacted by AST-masked tokens ($\tau < -0.1$). Our causal analysis yielded no signs of Transformers' performance being affected or guided by syntactic features, contradicting SOTA explainability findings.

6 CONCLUSION & FUTURE PLANS

Our negative causal effect results corroborate recent findings that show flaws when claiming that MLMs are *understanding* syntax rules of PLs. Such effects amplify the disparities between Natural and Programming languages, underscoring the need for tailored representations in deep learning architectures. These findings pave the way for future research to evaluate *semantic* capabilities in the form of *recursions, dead code, or code smells.* We also highlight the necessity to delve deeper into understanding why MLMs are more adept at predicting random masked tokens than syntax-based tokens. This tendency may be linked to the models' pre-training objectives, which frequently involve masking random tokens at a certain rate [10].

7 ACKNOWLEDGEMENTS

This research has been supported in part by the NSF CCF-2311469, CNS-2132281, CCF-2007246, and CCF-1955853. We also acknowledge support from Cisco Systems. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

REFERENCES

[1] 2023. WM-SEMERU/SyntaxEval. https://github.com/WM-SEMERU/SyntaxEval original-date: 2022-09-09T20:53:59Z.

[2] Toufique Ahmed, Dian Yu, Chengxuan Huang, Cathy Wang, et al. 2023. Towards Understanding What Code Language Models Learned. https://doi.org/10.48550/arXiv.2306.11943 arXiv:2306.11943 [cs].

[3] Vaishak Belle and Ioannis Papantonis. 2020. Principles and Practice of Explainable Machine Learning. *CoRR* abs/2009.11698 (2020). arXiv:2009.11698 https://arxiv.org/abs/2009.11698

[4] Marcel Bruch, Martin Monperrus, and Mira Mezini. [n. d.]. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (2009-08-24). ACM, 213–222. https://doi.org/10.1145/1595696.1595728

[5] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, et al. 2019. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2940179

[6] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, et al. [n. d.]. An Empirical Study on the Usage of Transformer Models for Code Completion. ([n. d.]), 1–1. https://doi.org/10.1109/TSE.2021.3128234

[7] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, et al. 2021. An Empirical Study on the Usage of Transformer Models for Code Completion. arXiv:cs.SE/2108.01585

[8] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Denys Poshyvanyk, et al. 2021. An Empirical Study on the Usage of BERT Models for Code Completion. *CoRR* abs/2103.07115 (2021). arXiv:2103.07115 https://arxiv.org/abs/2103.07115

[9] Colin Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, et al. [n. d.]. PyMT5: multi-mode translation of natural language and Python code with transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (2020). Association for Computational Linguistics, 9052–9065. https://doi.org/10.18653/v1/2020.emnlp-main.728

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. [n. d.]. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. https://doi.org/10.48550/arXiv.1810.04805 arXiv:1810.04805 [cs]

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:cs.CL/2002.08155

[12] Sangmok Han, David R. Wallace, and Robert C. Miller. [n. d.]. Code Completion from Abbreviated Input. In *2009 IEEE/ACM International Conference on Automated Software Engineering* (2009-11). IEEE, 332–343. https://doi.org/10.1109/ASE.2009.64

[13] Sangmok Han, David R. Wallace, and Robert C. Miller. [n. d.]. Code completion of multiple keywords from abbreviated input. 18, 3 ([n. d.]), 363–398. https://doi.org/10.1007/s10515-011-0083-2

[14] José Antonio Hernández López, Martin Weyssow, Jesús Sánchez Cuadrado, and Houari Sahraoui. 2022. AST-Probe: Recovering abstract syntax trees from hidden representations of pre-trained language models. *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Oct. 2022), 1–11. https://doi.org/10.1145/3551349.3556900 Conference Name: ASE '22: 37th IEEE/ACM International Conference on Automated Software Engineering ISBN: 9781450394758 Place: Rochester MI USA Publisher: ACM.

[15] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, et al. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*. 837–847. https://doi.org/10.1109/ICSE.2012.6227135

[16] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, et al. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. http://arxiv.org/abs/2308.10620 arXiv:2308.10620 [cs].

[17] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, et al. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *arXiv:1909.09436 [cs, stat]* (Sept. 2019). http://arxiv.org/abs/1909.09436 arXiv:1909.09436.

[18] Masahiro Kaneko, Masato Mita, Shun Kiyono, Jun Suzuki, et al. 2020. Encoder-Decoder Models Can Benefit from Pre-trained Masked Language Models in Grammatical Error Correction. arXiv:cs.CL/2005.00987

[19] Anjan Karmakar and Romain Robbes. 2023. INSPECT: Intrinsic and Systematic Probing Evaluation for Code Transformers. *IEEE Transactions on Software Engineering* (2023), 1–19. https://doi.org/10.1109/TSE.2023.3341624 Conference Name: IEEE Transactions on Software Engineering.

[20] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, et al. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv:cs.CL/1907.11692

[21] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, et al. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. https://doi.org/10.48550/arXiv.1907.11692 arXiv:1907.11692 [cs].

[22] Wei Ma, Mengjie Zhao, Xiaofei Xie, Qiang Hu, et al. 2023. Are Code Pre-trained Models Powerful to Learn Code Syntax and Semantics? https://doi.org/10.48550/arXiv.2212.10017 arXiv:2212.10017 [cs].

[23] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, et al. 2021. Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks. (2021), 336–347. https://doi.org/10.1109/icse43902.2021.00041

[24] Ahmad Haji Mohammadkhani and Hadi Hemmati. [n. d.]. Explainable AI for Pre-Trained Code Models: What Do They Learn? When They Do Not Work? ([n. d.]).

[25] David N. Palacio, Nathan Cooper, Alvaro Rodriguez, Kevin Moran, et al. [n. d.]. Toward a Theory of Causation for Interpreting Neural Code Models. https://doi.org/10.48550/arXiv.2302.03788 arXiv:2302.03788 [cs, stat]

[26] Judea Pearl. 2009. *Causality: models, reasoning, and inference*.

[27] Rafiqul Islam Rabin, Arjun Mukherjee, Omprakash Gnawali, and Mohammad Amin Alipour. [n. d.]. Towards Demystifying Dimensions of Source Code Embeddings. ([n. d.]), 29–38. ISBN: 9781450381253.

[28] Veselin Raychev, Martin Vechev, and Eran Yahav. [n. d.]. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014-06-09). ACM, 419–428. https://doi.org/10.1145/2594291.2594321

[29] Daniel Rodriguez-Cardenas, David N. Palacio, Dipin Khati, Henry Burke, et al. 2023. Benchmarking Causal Study to Interpret Large Language Models for Source Code. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 329–334. https://doi.org/10.1109/ICSME58846.2023.00040

[30] Amit Sharma, Vasilis Syrgkanis, Cheng Zhang, and Emre Kıcıman. 2021. DoWhy : Addressing Challenges in Expressing and Validating Causal Assumptions. (2021).

[31] P. K. Srimani and S. F. B. Nasir. 2007. *A Textbook on Automata Theory*. Foundation Books. https://doi.org/10.1017/UPO9788175968363

[32] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. [n. d.]. IntelliCode compose: code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020-11-08). ACM, 1433–1443. https://doi.org/10.1145/3368089.3417058

[33] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. [n. d.]. Pythia: AI-assisted Code Completion System. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2019-07-25). 2727–2735. https://doi.org/10.1145/3292500.3330699 arXiv:1912.00742 [cs]

[34] Sergey Troshin and Nadezhda Chirkova. 2022. Probing Pretrained Models of Source Codes. *Proceedings of the Fifth BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP* (2022), 371–383. https://doi.org/10.18653/v1/2022.blackboxnlp-1.31 Conference Name: Proceedings of the Fifth BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP Place: Abu Dhabi, United Arab Emirates (Hybrid) Publisher: Association for Computational Linguistics.

[35] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, et al. 2018. Deep Learning Similarities from Different Representations of Source Code. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 542–553.

[36] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, et al. 2021. Towards Automating Code Review Activities. In *43rd International Conference on Software Engineering, ICSE'21*. https://arxiv.org/abs/2101.02518

[37] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, et al. [n. d.]. What Do They Capture? – A Structural Analysis of Pre-Trained Language Models for Source Code. https://doi.org/10.48550/arXiv.2202.06840 arXiv:2202.06840 [cs]

[38] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, et al. 2022. A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research. *ACM Transactions on Software Engineering and Methodology* 31, 2 (March 2022), 32:1–32:58. https://doi.org/10.1145/3485275

[39] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, et al. 2020. On Learning Meaningful Assert Statements for Unit Test Cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1398–1409. https://doi.org/10.1145/3377811.3380429

[40] Martin White, Michele Tufano, Matías Martínez, Martin Monperrus, et al. 2019. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 479–490. https://doi.org/10.1109/SANER.2019.8668043

[41] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 87–98.

[42] Martin White, Christopher Vendome, Mario Linares-Vasquez, and Denys Poshyvanyk. [n. d.]. Toward Deep Learning Software Repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories* (2015-05). IEEE, 334–345. https://doi.org/10.1109/MSR.2015.38