COMP 599 Final Project: What do code-based Transformer models learn?

Mika Desblancs-Patel
Philosophy Department

Hector Leos
Cognitive Science Department

Adam Weiss
Computer Science Department

Abstract

With the advent of state-of-the-art large-scale pre-trained Transformer-based models in various domains of Natural Language Processing, recent work has focused on understanding what about natural language these models learn. While some studies show they learn grammatical structures, others cast doubt over such claims. In addition to natural language tasks, transformer models, such as CodeBERT have also recently been used in many tasks involving code. However, little work has been done to understand whether Transformers are able to understand key structures related to code. In this paper, we provide novel evidence that state-of-the-art Code-Clone detection models are largely invariant to random word-order permutations (ie. they assign the same labels to code pairs that have been permuted and those which haven't). We provide preliminary empirical evaluation of this phenomenon. Furthermore, we also find evidence that Transformers are capable of capturing important syntactic structures, as previously shown in NL models. Syntax structures such as corefence are captured to a great extent. We discuss the implications of these findings in the effort of understanding what these models are learning at large.

1 Introduction

Transformer-based models (Devlin et al.) have achieved impressive results in many Natural Language Processing related tasks, playing integral parts in most state-of-the-art models such as natural-language inference, sentiment analysis, and language modelling etc. Due to their success working with natural language, researchers have created Transformer-based models for tasks related to code, such as code to code (clone detection, defect detection, code completion), text to code (natural language code search, text-to-code generation), code to text in code summarization and text-text in document translation (Lu et al. 2021).

Just as many other Deep Learning models, Transformers are hard to understand and figuring out what exactly they are learning is difficult. Recently, researchers have tried to analyze what Transformer models learn about language through various methods (Manning et al., 2020; Voita et al., 2019). They have argued that Transformer models are capable of learning certain syntactic structures in code and that their subsequent high performance on many language modelling tasks is in part due to learning these structures. Others, have recently cast doubt over these claims (Sinha et al. 2020) but finding that Transformer models seem to perform similarly on tasks requiring hypothesized semantic understanding whether the input is shuffled or not. Given that humans struggle to understand shuffled sentences, the authors conclude that since Transformer models seem token order invariant, Transformer models cannot learn too much language structure.

Transformer models are also used in code-related tasks and have achieved impressive results. However, no work has been done on understanding what about code they are in fact learning. Programming languages and code are rigorously defined languages with clear syntactic structures and little flexibility in terms of token order and structure. If we are to claim Transformers can learn complex structures related to natural language, then we would expect Transformer models to learn linguistic structures in programming languages with a higher degree of certainty that with natural languages since it is more clearly defined. In this paper, we explore what structures Transformer-based models trained to encode code actually learn. We present two experiments, the first looks at attention heads at various layers in the Transformer and the other investigates a model's performance on permutated inputs to determine how important word order is to model performance.

2 Related Work

2.1 Transformer models

Transformers are a type of neural network architecture developed to perform neural machine translation (Vaswani et al, 2017). Similarly to previous sequence translation models such as RNNs, LSTMs (Hochreiter, 1997), gated recurrent networks (Chung et al., 2014), Transformers can be used as both encoders and decoders. What makes it different from the rest is its use of self-attention: a mechanism by which the model attends to different positions of the input sequence to compute a contextual representation of all the words in a sequence. By running through an attention mechanism several times and getting several different attention heads (i.e. multihead attention), the model can jointly attend to information from different representation subspaces at different positions, and is capable of learning contextual relations between words in a text. Concretely, this means different heads can attend to parts of the sequence differently and capture different types of information. For example, the case of natural language, it seems that different heads pay special attention to positional, syntactic or rare word information (Voita et al. 2019).

BERT (Bidirectional Encoder Representation from Transformers) is a state-of-the-art language representation model based on the Transformer architecture which has gained prominence by dominating multiple key NLP benchmarks. (Devlin et al., 2019). Its architecture is almost identical to the original Transformer model in that it does not use recurrence but rather attention and feed-forward layers. Unlike its predecessor, however, BERT is only an encoder whose purpose is to find contextual word representations which can be used for numerous downstream tasks.

2.2 Attention head analyses

To better understand why Transformer models are so effective at solving their tasks, there is a need to analyse their learning mechanisms and acquired representations. There have been efforts to examine the kind of information learned by multi-head attention by averaging the attention weights over all heads at a given position (Voita et al., 2018), or by only considering the the maximum attention weights (Tang et al., 2018). The problem with these methods is that they don't explicitly take into account the varying importance of different heads. (Voita et al. 2019) propose a pruning mechanism by which irrelevant heads are removed. This method allows them to evaluate the contribution made by individual attention heads in the encoder to the overall performance of the model.

Manning et al. (2019) quantitatively analyse the correspondence between attention heads and linguistic phenomena in BERT by observing the behavior of attention heads on annotated data. More specifically, they treat each attention head as a classifier for every syntactic relation in language, and they calculate the precision across the whole data set. The authors found that Bert effectively learns structure in language. For example, an attention head identifies direct object relations with 87% accuracy, and another one captures noun premodifiers with 94% accuracy.

2.3 Unnatural Language Inference

Casting doubt on previous work arguing that Transformer models learn syntactic structures, researchers have found that state-of-the-art Transformer bades models appear to be largely order invariant (Sinha et al. 2021). First they evaluate various models on multiple natural language inference tasks (NLI). Then, for each hypothesis-entailment pair, they use a permutation function to create 100 different permutations for both the hypothesis and entailment sentence subject to the constraint that no word retains its original position. They find that 98.5% of the list of permutations contain at least one example where the model makes the correct decision using a permuted pair and 83.6% where the model predicts correctly on $2/3$ permutations in the list. In many cases, the model initially makes a wrong prediction before making a correct one on the permutated pairs. This casts doubt on how important syntactic structure learning is important for Transformer-based models because word order is one of the most important structure for humans who struggle to understand shuffled sentences.

2.4 Code models

Transformers were developed to work with natural language, however since natural language and code both share structure, namely they are both easily decomposed into sequences of tokens, many teams have successfully used the architecture for code related tasks. One such model is CodeBERT (Feng et al. 2020), a BERT style model for code instead of language. It was developed using exactly the same architecture as RoBERTa-base (Liu et al., 2019). CodeBERT was pre-trained using natural language-code pairs (bimodal data) by pairing code functions with its documentation along with code (unimodal data) using code functions alone. It was pre-trained using two training tasks: masked language modelling (MLM) on binomial data and replaced token detection (RTD) on both bimodal and unimodal data. The MLM task involves masking tokens in the code and natural language sequence, following (Devlin et al. 2018). The training objective is to predict the masked tokens. Given a series of tokens in natural language and code where certain tokens have been replaced by plausible alternatives, the RTD objective is to train the model to discriminate between original and replacement words. Since CodeBERT is an encoder model, it is easily available for downstream tasks. In this paper we follow the author's fine-tuning procedure for Code-Clone Detection (Feng et al. 2020). We chose this model due to the ease of use and computational restrictions which are detailed in the appendix.

Transformers have also been used for code translation tasks. TransCoder (Baptisite et al. 2021) is a seq2seq Transformer model that translates code from one programming language to another, and which has been trained in an unsupervised manner with 3 paradigms. The first is akin to a MLM in that code tokens are masked and the model is tasked with predicting them. This does not involve translation but allows for representations of code to be trained. After this a step (called denoising) auto-encoding can be applied, which essentially means that the decoder is trained by applying a standard sequence prediction task with missing tokens. Then a final step of back translation is used to train the model. In our paper, we use the Transcoder encoder to analyze what Transformer heads are able to learn about code.

While we would have originally preferred to keep our analysis to one Transformer-based model, our original idea involving Transcoder turned out to be unfeasible given the time-constraints and computational limitations. Code-clone detection with CodeBERT was our plan B. However, given that both CodeBERT and Transcoder are Transformer-based models, we believe our results are still indicative of general Transformer behaviour.

2.5 Code Clone Detection

(Sinha et al., 2020) use a binary classification task (NLI) to test the effects of permutations on model performance, code-related natural language tasks also has similar tasks. Code clone detection is a binary classification task where the goal is to detect if two differing pieces of code actually correspond to the same thing. Given a pair code fragments (C_{i1}, C_{i2}), the goal of the task is to predict a label y_i indicating whether or not they are clones. Researchers agree that we can define two snippets of code as clones by referring to the following exclusive list of clone types:

Type-1 : Syntactically identical code snippets, except for differences in white space, layout and comments.

Type-2 : Syntactically identical code snippets, except for differences in identifier names, literal values, white space, layout and comments.

Type-3 : Syntactically similar code snippets that differ at the statement level. Snippets have statements added, modified and/or removed with respect to each other.

Type-4 : Syntactically dissimilar code snippets that implement the same functionality.

This has a wide range of applications including education where it can be used to detect plagiarism and in industry where it can assist in refactoring by pointing out redundant code. We chose code clone detection as our task because entailment is not well defined for code, but code clone detection is well-defined binary classification task. For our project, we use the BigCloneBench dataset (Svajlenko et al. 2015) where snippets of code are tagged as clones or not depending on whether they implement the same functionality. It also contains all the different clone types described previously. We believe Clone-detection is the closest code-related natural language task to provide a good test for whether a model understands the semantic similarity between two code. Here, however, semantic similarity is understood as functional similarity.

3 Our Approach

3.1 Attention-head Analysis

In order to understand what syntactic structure in code the model is learning, we will focus on the various attention heads of our Transformer-based encoder. For this purpose, we'll follow the same methodology as Manning et al. (2019), who demonstrated that modern deep contextual language models can learn major aspects of natural language structure through self-supervision. We'll analyze the structural phenomena these models uncover in code, and we'll compare our results with those of Manning (using natural language). For example, Python's keywords 'or' and 'and' have the same syntactic functionality as their natural language counterparts, so comparison across models should be easy. However, there are structural constraints in code that are not present in language. We'll assess the similarities and differences in these learned representations in order to contrast the syntactic structures of natural language and code.

3.2 Permutation Experiment

Similar to previous work (Sinha et al.2021), we also hypothesize that if a model understands a piece of code, then, our model will perform poorly when perdicting over that equivalent bit of code such that it's tokens are permuted since no longer compiles, is unable to produce a sequence of actions and no longer performs its original function. It becomes functionless. Then, a model trained on the non-permuted sentence should no longer be able to produce meaningful results on the permuted example. In the case of the code-clone detection task, if both bits of code in the pair are permuted, we hypothesize that our model barely surpasses a base random prediction.

In order to test how well a model deals with permuted pairs of code, we construct a permuted dataset. For a given dataset D having splits D_{train} and D_{test}, we first reproduce the CodeBERT clone detection experiment using D_{train} for training and D_{test} for testing. Then, we construct a randomized version of Dtest where for each example (c_{i1}, c_{i2}, l_i), where c_{ij} is the j^{th} code in i^{th} pair and l_i the label of pair i. We use a permutation operator f which will return a new list $(\hat{C_{i1}}, \hat{C_{i2}}, L)$ where $\hat{C_i j}$ is now a list of permutations of the code c_{ij}. The permutations are such that no token retains its same position between the original code and the permuted pair. For each pair(c_{i1}, c_{i2}, l_i), we evaluate our model's performance on both the un-permutated and permutated pairs. Importantly, we don't retrain the model when testing it on the permutated pairs.

For our paper, we sample $1,000$ datapoints from the BigCloneBench data test set, 500 positive pairs and 500 negative pairs. Due to computational constraints, we were not able to run our experiment on a bigger sample. For each snippet of code, we create 50 permuted versions in order to create 50 pairs of code on which we predict whether they are clones.

3.3 Dataset

Microsoft provided a code clone detection dataset as part of their package CodeXGLUE (Lu et al. 2021) along with a pipeline to fine-tune a model on it, and hence we used this dataset as our base datset.

BigClone-Bench is a data-set for code clone detection (Svajlenko et al. 2014) containing $1,731,860$ pairs, a training set of $901,028$ pairs, a development set of $415,416$ pairs and a test set of $415,416$ pairs. The state of the art can achieve Precision and Recall of .96 and an F1 score of .95, however those models are Graph Neural Network Based and not related to language models, and the CodeBERT model we use as a baseline achieves precision, recall, and F1 of .95, .93, and .94 respectively. This dataset came with an easy to use fine-tuning and evaluation pipeline, and was selected for this reason.

3.4 Model selection

We decide to use the CodeBERT model which we fine-tuned on the BigCloneData clone detection dataset. We fine-tuned and evaluated on a machine with 85GB of RAM and 1 A100 GPU. We fined tuned for 2 epochs with the Adam optimizer, a batch size of 16, a learning rate of $5e - 5$, and gradients clipped to 1.0. All models are built in PyTorch. We ran the experiment for 3 runs as everything is deterministic, so more runs would not yield better data. For the attention head analysis we use the pre-trained Transcoder encoder with no extra training or fine-tuning.

4 Evaluation metrics

4.1 Attention heads analysis

In this project we will employ Manning et al.'s method for the attention head analysis. More specifically, let $l(w_i, w_j)$ be the assigned label to a pair of words, with value 1 if a particular linguistic relationship exists between words w_i and w_j, 0 otherwise. $\alpha(w, h)$ denotes the attention distribution of head h, when the model is run over the sequence of words $w = [w_1, ..., w_n]$. For each position $1 \leq j \leq n$, the most-attended-to-word is defined as $w_{argmax_i \alpha(w,h)^j_i}$. Then, we defined $S_l(w) = j : \sum_n^{i=1} l(w_i, w_j) > 0$ as the subset of the input expressing the annotated relationship. The precision score for each attention head is computed as:

$$precision(h) = \frac{1}{N}\sum_{w \in corpus}\sum_{j \in S_l(w)} l(w_{argmax_i \alpha(w,h)_i^j}, w_j)$$

where N is the total number of words in the corpus expressing the relationship. This score is a way of evaluating the attention head as a simple classifier that predicts the presence of the linguistic relationship of interest.

For example, let's assume we were interested in investigating whether attention heads captured direct object relationships, and that our corpus consisted on the following two sentences: "Larry saw her yesterday" and "Mike gave me a gift". Then, only $"her", "me" \in corpus$ would be relevant, since $S_l("her") = \{"Larry"\}$ and $S_l("me") = \{"Mike"\}$. (For the rest of the words, these subsets would be empty, since they don't partake in a direct object relation). Assuming that for $head_i$, $"her"$ was the most attended word of $"Larry"$ in sentence one, but $"me"$ was not the most attended word of $"Mike"$ in sentence two, then $precision(head_i) = 0.5$.

We required annotated code to utilize Manning's method in our project. However, since no one has investigated the structural phenomena of code, annotated code corpora are not available. Therefore, we came up with the following syntactic relations that we hypothesized could be found in Java code:

- **VarAss:** the relation between a variable and the value it's assigned to
- **DataType:** the relation between a variable and its data type (e.g. String x)
- **RetType:** the relation between the returned value of a function and the function's return type (indicated in its header)
- **CorefVar:** variable coreference
- **CorefKey:** coreference between keywords such as "install", "for", "try", data types, etc.
- **CorefPunc:** coreference between punctuation marks such as ; and :
- **CorefExp:** coreference between whole expressions (n>3) such as `System.out.println(...)`
- **ParMt:** relation between an opening parenthesis and a matching closing one
- **BrackMt:** relation between an opening curly bracket and a matching closing one

We also used an attention head visualizer (BertViz; see Vig, 2019) to obtain visual representations of TransCoder's attention heads. This top-down approach permitted us to have an idea of more non-intuitive kinds of relationships in code captured by the attention heads. These are the relationships we further identified:

- **PrevW:** relation between a word and its predecessor
- **NextW:** relation between word and the next one
- **LoneMt:** a word that only appeared once in a given code snippet (i.e. that didn't have a coreferent) matching itself
- **AllPar**, **AllSep**, and **AllSemic:** the percentage of words in a sentence looking at a specific token (closing parenthesis, separator token, and semicolon respectively)

For each head we treat it as a classifier and measure the precision with which it classifies a syntactic relation correctly, the results of this are visible in the results section where we have a table denoting which attention head had the highest precision for the prediction of various relations.

4.2 Permutation Acceptance Metrics

Given (c_{i1}, c_{i2}, y) containing a two snippets of code and a label indicating whether the pair is a clone or not, our model will predict a prediction $p = \hat{y}$. A True Positive (TP) occurs when the $y = \hat{y} = 1$,a True Positive (TN) occurs when $y = \hat{y} = 0$, a False Positive (FP) occurs when the label is $y = 0, \hat{y} = 1$, and a False Negative (FN) occurs when $y = 1, \hat{y} = 0$. Let TP, FP, TN, FN denote the counts of each, then true-positive rate (TPR), false-positive rate (FPR), true-negative rate (TNR) and false-negative rate (FNR) are defined by:

$$TPR = \frac{TP}{n}$$
$$FPR = \frac{FP}{n}$$
$$TNR = \frac{TN}{n}$$
$$FNR = \frac{FN}{n}$$

5 Results

The results for the attention head analysis conducted on TransCoder are presented in the following table:

Attention head precision scores		
Relation	Precision (%)	Head
VarAss	80.4	0-1
DataType	36.3	0-2
RetTyp	< 1.0	-
CorefVar	97.8	0-5
CorefKey	91.1	0-5
CorefPunc	64.4	1-4
CoredExp	93.9	1-4
ParMt	59.1	1-5
BrackMt	99.6	0-0
PrevW	100	0-2
NextW	91	3-3
LoneMt	92.1	3-1
AllPar	82.1	5-4
AllSep	39.5	1-2
AllSemic	22.32	3-5

Figure 1 depicts the syntactic relations captured by the 48 attention heads of TransCoder. It is worth noticing that head 1-4 seems to be capturing coreference relations very well, as well as head 3-1. Figure 2 shows some examples.

In Table 1, we get the results for the permutation experiment. In the first column we get the types of decision that our model can make about a datapoint. In the subsequent columns we get the TPR, FPR, FNR and TNR for the decisions made by our model for every decision type. Note that cells filled with $None$ indicate that there can not be a positive number in this cell. For example, among the TP datapoints, we can only get a FPR of 0 since we cannot falsely predict a point as positive if it is positive. We find the model has an F1 score of 98% score higher than the model presented in the original paper which had an F1 score of 94% (Feng et al. 2020). We find the model discriminates extremely well between the positive and negative labels when the code pairs are unshuffled. Surprisingly, the model performs extremely well on the shuffled pairs as well. With a TPR of 81% among the TP points, TNR of 0.99% among the TN points and FNR of 0.99% among the Fn points, predictions made on the shuffled pairs is in strong agreement with our model's choices.

Decision	TPR	FPR	FNR	TNR
$TP_{p=0.49}$	0.81	None	0.19	None
$FP_{p=0}$	None	0	None	0
$TN_{p=0.49}$	None	0.001	None	0.99
$FN_{p=0.02}$	0.01	None	0.99	None

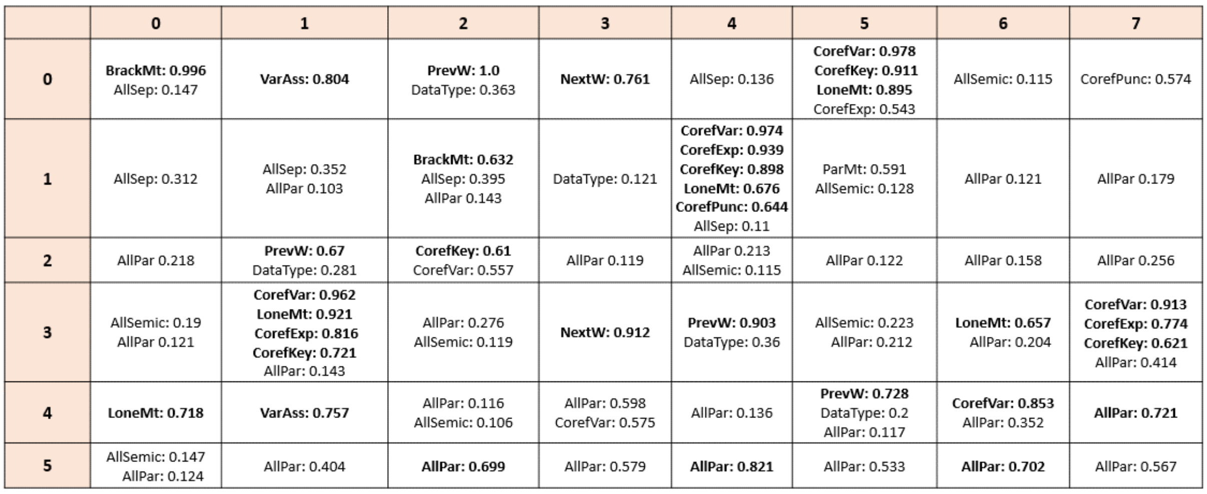
Table 1: Results of the permutation experiment with n the number of datapoints with a given type, TPR, FPR, FNR and TNR as defined before, $None$ being cells where no value is possible, and p the probability of the point type.

6 Discussion

6.1 Attention head analysis

Our attention head analysis has shown that Transformer models are able to capture syntactic structure, not only in natural language as demonstrated by Manning et al. (2020), but in code as well. Some comparisons could be made. For instance, Manning et al. found that words most attend to their coreferent 65.1% of the time, whereas in code, variables attend to their coreferent 97.8% of the time, keywords 91.1% of the time, whole expressions 93.9% and punctuation marks 64.4%. This could be explained by the fact that coreferents in code don't change names as in natural language (i.e. variable x will always be named x, whereas "Maria" can be the coreferent for "her"), which might make it easier for the Transformer to identify coreference relations. Direct objects in NL, which obtained a 86.8% accuracy in Manning, can loosely be compared to variable assignation in code (80.4% accuracy). A phenomenon particular to Java code is the structure given by curly brackets: we found that head 0-0 matches opening to closing curly brackets 99.6% of the time, which indicates that hierarchical structure in code is captured early on in TransCoder's encoder. A relation that this model failed to capture was that of the returned value of a function to its return type, indicated in the function's header (less than 10% accuracy). These attention heads were also not sensitive to the relation between a variable and its data type, the most successful one obtaining a 36.3% accuracy. Thus, Transformers' attention heads seem to perform badly when it comes to identifying data type information in code.

These results are important because they show that Transformer models capture structure in data different than natural language. Furthermore, they provide further evidence to Voita et al.'s (2018) hy-



	0	1	2	3	4	5	6	7
0	**BrackMt: 0.996** AllSep: 0.147	**VarAss: 0.804**	**PrevW: 1.0** DataType: 0.363	**NextW: 0.761**	AllSep: 0.136	**CorefVar: 0.978** **CorefKey: 0.911** **LoneMt: 0.895** CorefExp: 0.543	AllSemic: 0.115	CorefPunc: 0.574
1	AllSep: 0.312	AllSep: 0.352 AllPar 0.103	**BrackMt: 0.632** AllSep: 0.395 AllPar 0.143	DataType: 0.121	**CorefVar: 0.974** **CorefExp: 0.939** **CorefKey: 0.898** **LoneMt: 0.676** **CorefPunc: 0.644** AllSep: 0.11	ParMt: 0.591 AllSemic: 0.128	AllPar 0.121	AllPar 0.179
2	AllPar 0.218	**PrevW: 0.67** DataType: 0.281	**CorefKey: 0.61** CorefVar: 0.557	AllPar 0.119	AllPar 0.213 AllSemic: 0.115	AllPar 0.122	AllPar 0.158	AllPar 0.256
3	AllSemic: 0.19 AllPar 0.121	**CorefVar: 0.962** **LoneMt: 0.921** **CorefExp: 0.816** **CorefKey: 0.721** AllPar: 0.143	AllPar: 0.276 AllSemic: 0.119	**NextW: 0.912**	**PrevW: 0.903** DataType: 0.36	AllSemic: 0.223 AllPar: 0.212	**LoneMt: 0.657** AllPar: 0.204	**CorefVar: 0.913** **CorefExp: 0.774** **CorefKey: 0.621** AllPar: 0.414
4	**LoneMt: 0.718**	**VarAss: 0.757**	AllPar: 0.116 AllSemic: 0.106	AllPar: 0.598 CorefVar: 0.575	AllPar: 0.136	**PrevW: 0.728** DataType: 0.2 AllPar: 0.117	**CorefVar: 0.853** AllPar: 0.352	**AllPar: 0.721**
5	AllSemic: 0.147 AllPar: 0.124	AllPar: 0.404	**AllPar: 0.699**	AllPar: 0.579	**AllPar: 0.821**	AllPar: 0.533	**AllPar: 0.702**	AllPar: 0.567

Figure 1: Syntactic relations in code found in TransCoder's attention heads. Rows represent layers, columns represent heads.

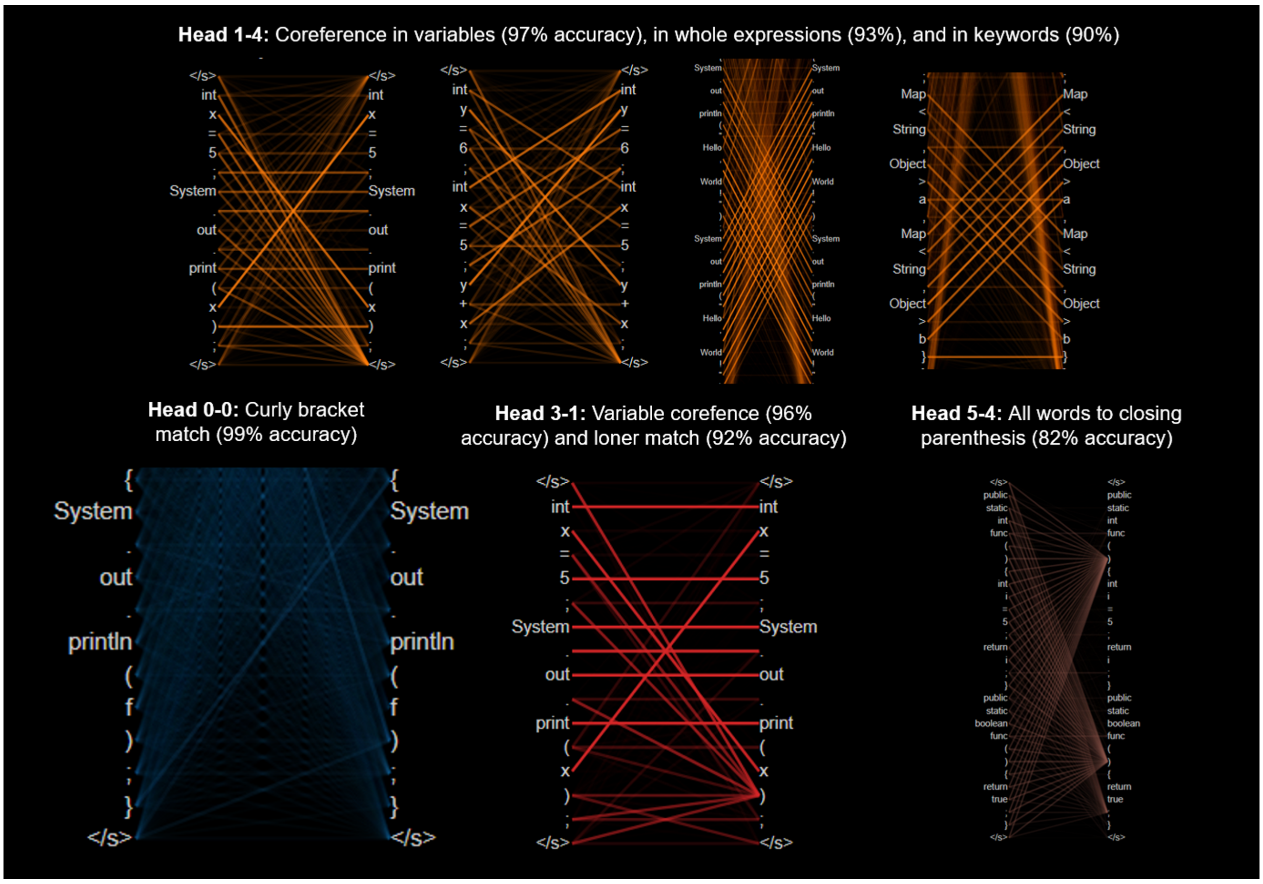


Figure 2: The attention heads in TransCoder that appear to be the most sensitive to code syntactic structures, such as coreference, curly brackets match and attention to closing parentheses.

pothesis that only some certain heads are important for the performance of these models. As can been seen in Figure 1, attention heads 0-5, 1-4, 3-1, 3-7 seem to be capturing the most syntactic relations when compared to the rest. We could predict that if one were to conduct Voita et al.'s pruning method to dispense with the unnecessary heads, these four would be part of the few remaining. Further work could try to replicate our findings with other types of code-based Transformers and other programming languages, to see whether these results are particular to the particular task at hand (i.e. code translation) or to Java syntax. Additionally, the decoder's attention heads could be analysed and integrated to the present encoder analysis, in order to obtain a more comprehensive understanding of the Transformer's learning mechanisms. Finally, more work has to be done to understand the extent to which attention heads learning impacts model performance.

6.2 Code clone detection

The extremely high level of agreement between our model's decisions on the permuted examples and unpermuted examples is highly surprising. We would expect our model to perform terribly in the TPR of the TP points. That the TPR is not high on the TP points as the TNR on the TN matches our intuitions: given that code is scramble and thus both of their functions are no longer identical (in fact they don't function at all), our model should more easily predict that two bits of code are not clones because they in fact don't have the same function. However, a TPR of 81% for the TP points is still very high. This indicates that the CodeBERT model seems to be using almost exclusively syntactic similarity in order to make predictions.

This seems to contradict the claims found in our discussion on attention heads which argued that the Transcoder encoder is learning syntactic structure in code. With these results, there could be two possible explanations: either CodeBERT does not capture semantic similarity and the Transcoder Transformer encoder does, or, token order is a structure that Transformer models are sensitive to. The former interpretation is possible, and future studies would have to look into this. However, since both are Transformer-based, we would expect similar findings for most Transformer-based models. The latter interpretation would agree with the discrepancy between the claims found in (Manning et al.) and (Sinha et al. 2021) whereby Transformer heads seemed to learn syntactic structure in natural language but were insensitive to word order for natural language inference tasks.

Finally, the high scores we get in reproducing the experiment can be due to the small sample from the test set that we use due to computational constraints. It is possible that our sample contains pairs that our model can deal with easily. It is also possible that our sample contains many pairs with strong syntactic similarity, hence explaining why the permutated pairs are no problem for our model.

7 Future Work & Conclusion

In conclusion, like natural language models, Transformer models for code were found to capture different syntactic relations in code. In particular, coreference relations were effectively learned by different attention heads, which might indicate that coreference is an important feature for code translation. Our results complement those of Manning et al., and validate the observations done by Voita et al. regarding the importance of only certain attention heads in Transformer models. This work contributes to the effort of understanding the machinery behind state-of-the-art NLP models, and opens the possibility of integrating other kinds of data (i.e. code) into model learning analyses.

In addition like natural language models, code models do not degrade significantly on downstream tasks when the input is shuffled. This shows that code models, like language models, although they learn some syntactic structure, ultimately rely on statistical properties rather than semantic ones in making a prediction.

The model performance indicates that the model likely learns statistical properties of code rather than actual semantics, and future work could include some form of contrastive training as was presented in class. Future work could also include a departure from natural language models for code such as using graph models on syntax trees.Another potential way to address the shortcomings of transformers may be to add a fine tuning step where grammatical structures are labelled on a set of inputs and a model is penalized according how strongly a head that picks up on these structures exists.

Appendix

7.1 Division of Labor

Mika, Héctor, and Adam all contributed equally in volume overall and in writing this report. Héctor adapted BertViz to the TransCoder model, came up with the syntactic relations in code and labelled the data according to these relations, and conducted the attention head analysis. Mika designed the data shuffling pipeline, code token order experiment and worked with Adam on getting compute resources such as Google Cloud. Adam did a lot of engineering work including wrangling the pre-existing pipelines into a useful form, figuring out efficient ways to handle the large volume of data, and working on the model and dataset design/selection for code clone detection.

7.2 Code and data

Please see our Github repository at https://github.com/mika-jpd/Comp-599-Final-Project. See CodeXGLUE at https://github.com/microsoft/CodeXGLUE.

7.3 Limits due to computational constraints

We were highly impacted computational constraints related to memory size in the development of our own classifier atop CodeBERT embeddings and therefore had to use a pre-exisiting classifier. Our attempt our own classifier can be found as codebertsimilar.ipynb in our repo. In addition we needed the extra GPU memory provided by Google Cloud to run our experiments with a pre-made model. This was not a skill any of us had, and required technical support from Google to even turn the server on (Google did not provide any scientific or experimental input), this contributed to our delays. In summary, the scope of our project was severely limited due to memory constraints and general compute constraints.

8 References

K. Clark, U. Khandelwal, O. Levy, C. D. Manning, “What does BERT look at? An analysis of BERT’s attention” in Proceedings of the Second BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP, T. Linzen, G. Chrupała, Y. Belinkov, D. Hupkes, Eds. (Association for Computational Linguistics, Stroudsburg PA, 2019), pp. 276–286.

Christopher D. Manning, Kevin Clark, John Hewitt, Urvashi Khandelwal, Omer Levy, "Emergent linguistic structure in artificial neural networks trained by self-supervision" in Proceedings of the National Academy of Sciences Dec 2020, 117 (48) 30046-30054; DOI:

10.1073/pnas.1907367117 alphacode, Competition-Level Code Generation with AlphaCode, Li, Yujia and Choi, David and Chung, Junyoung and Kushman, Nate and Schrittwieser, Julian and Leblond, Rémi and Eccles, Tom and Keeling, James and Gimeno, Felix and Dal Lago, Agustin and Hubert, Thomas and Choy, Peter and de Masson d’Autume, Cyprien and Babuschkin, Igor and Chen, Xinyun and Huang, Po-Sen and Welbl, Johannes and Gowal, Sven and Cherepanov, Alexey and Molloy, James and Mankowitz, Daniel and Sutherland Robson, Esme and Kohli, Pushmeet and de Freitas, Nando and Kavukcuoglu, Koray and Vinyals, Oriol,2022,Feb

R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, et al. Project CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks. arXiv preprint arXiv:2105.12655, 2021.

Ondřej Bojar, Christian Buck, Christian Federmann, Barry Haddow, Philipp Koehn, Johannes Leveling, Christof Monz, Pavel Pecina, Matt Post, Herve Saint-Amand, Radu Soricut, Lucia Specia, and Aleš Tamchyna. 2014. Findings of the 2014 Workshop on Statistical Machine Translation. In Proceedings of the Ninth Workshop on Statistical Machine Translation, pages 12–58, Baltimore, Maryland, USA. Association for Computational Linguistics.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention is all you need. InAdvances in Neural Information Processing Systems(pp. 5998-6008).

Sutskever, I., Vinyals, O., Le, Q. V. (2014). Sequence to sequence learning with neural networks. InAdvances in neural information processing systems(pp. 3104-3112).

Bahdanau, D., Cho, K., Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate.arXiv preprint arXiv:1409.0473.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Advances in neural information processing systems, pp. 5998–6008, 2017.

Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. ¨ Neural Comput., 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL http://dx. doi.org/10.1162/neco.1997.9.8.1735.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. NAACL, 2019. URL https://arxiv. org/pdf/1810.04805.pdf.

Wu, Lijun, et al. "R-drop: regularized dropout for neural networks." Advances in Neural Information Processing Systems 34 (2021).

Zhu, J., Xia, Y., Wu, L., He, D., Qin, T., Zhou, W., ... Liu, T. Y. (2020). Incorporating bert into neural machine translation. arXiv preprint arXiv:2002.06823.
journals/corr/abs-2102-04664,Shuai Lu and Daya Guo and Shuo Ren and Junjie Huang and Alexey Svyatkovskiy and Ambrosio Blanco and Colin B. Clement and Dawn Drain and Daxin Jiang and Duyu Tang and Ge Li and Lidong Zhou and Linjun Shou and Long Zhou and Michele Tufano and Ming Gong and Ming Zhou and Nan Duan and Neel Sundaresan and Shao Kun Deng and Shengyu Fu and Shujie Liu,CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understand Generation,CoRR,abs/2102.04664,2021
Feng, Zhangyin, et al. "Codebert: A pre-trained model for programming and natural languages." arXiv preprint arXiv:2002.08155 (2020).
Towards a big data curated benchmark of inter-project code clones,Svajlenko, Jeffrey and Islam, Judith F and Keivanloo, Iman and Roy, Chanchal K and Mid Mamun,2014 IEEE International Conference on Software Maintenance and Evolution,476–480,year=2014.

Unsupervised translation of programming languages,Roziere, Baptiste and Lachaux, Marie-Anne andChanussot, Lowik and Lample, Guillaume,Advances in Neural Information Processing Systems,33,2020

Svajlenko, Jeffrey Roy, Chanchal. (2016). BigCloneEval: A Clone Detection Tool Evaluation Framework with BigCloneBench. 596-600. 10.1109/ICSME.2016.62.