

Empirical Study of Transformers for Source Code

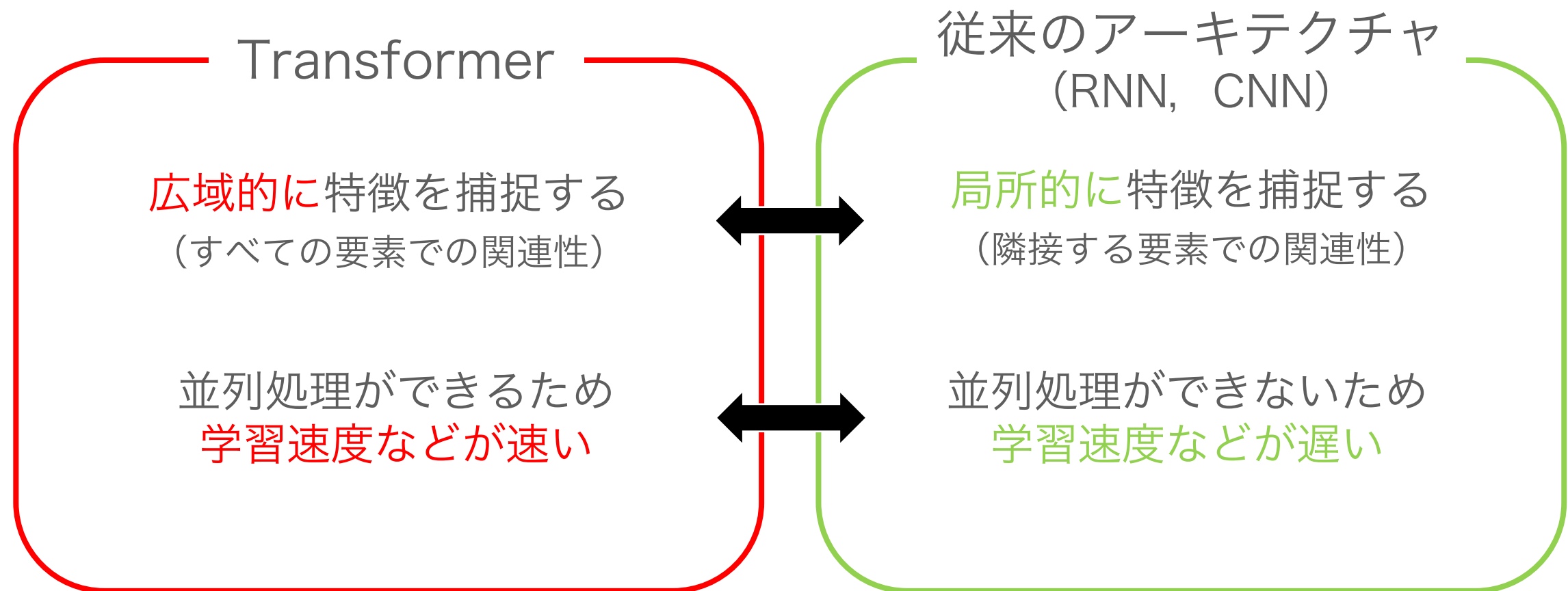
Nadezhda Chirkova
HSE University
Moscow, Russia
nchirkova@hse.ru

Sergey Troshin
HSE University
Moscow, Russia
stroshin@hse.ru

In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021), pp 703--715.

Transformer

- いま主流のアーキテクチャでさまざまな処理タスクにおいて利用
 - コード補完^[20], コード翻訳^[21, 34], バグ修正^[14]
- 従来の深層学習アーキテクチャに比べて優れた性能



Transformerとプログラムコード

- Transformerは学習時に各要素の構造情報が必要
 - 通常, positional encoding (embedding) で補完
- これまでの研究ではコードの木構造に着目 ^[14, 20, 32]

既存研究の課題

- 対象としてるタスクが異なる
- コードの構造情報を効果的に捉える方法が不明
 - 比較対象が標準のTransformerモデル
- 木構造にのみ着目し他の構造情報を利用していない

目的

- コード処理タスクにおけるTransformerモデルの利用について調査
 - SE*タスクごとに構造情報の最も良い利用方法とは？
 - Transformerモデルはコードの構文を捉えるのに適しているのか？

SE*：ソフトウェアエンジニアリング

基本データの成形

1. 入力されたコードを抽象構文木に変換

(例) 値の代入

- NameStore : 変数の用意
- NameLoad : 変数呼び出し
- Index : 添字
- Assign : 値の割り当て (代入)
- SubscriptLoad : 参照呼び出し

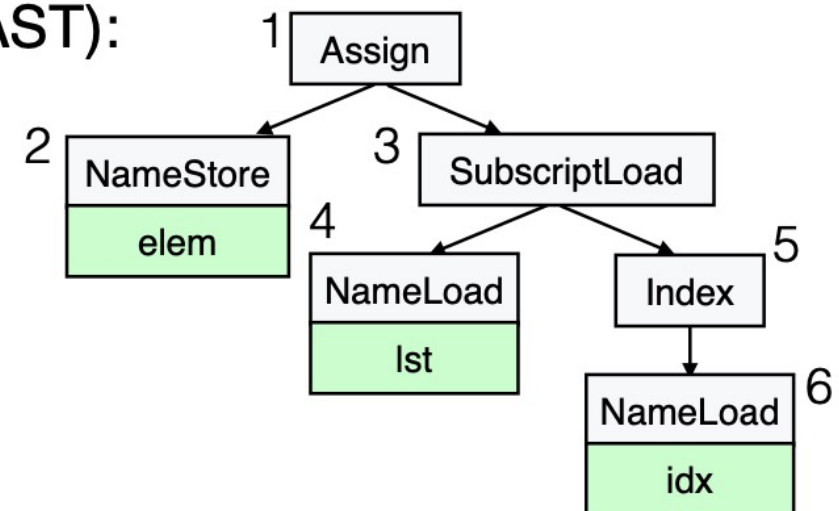
2. 抽象構文木を深さ優先探索で整列

(a) 入力されるコード

(a) Code: `elem = lst[idx]`

(b) AST : 抽象構文木

(b) Abstract syntax tree (AST):



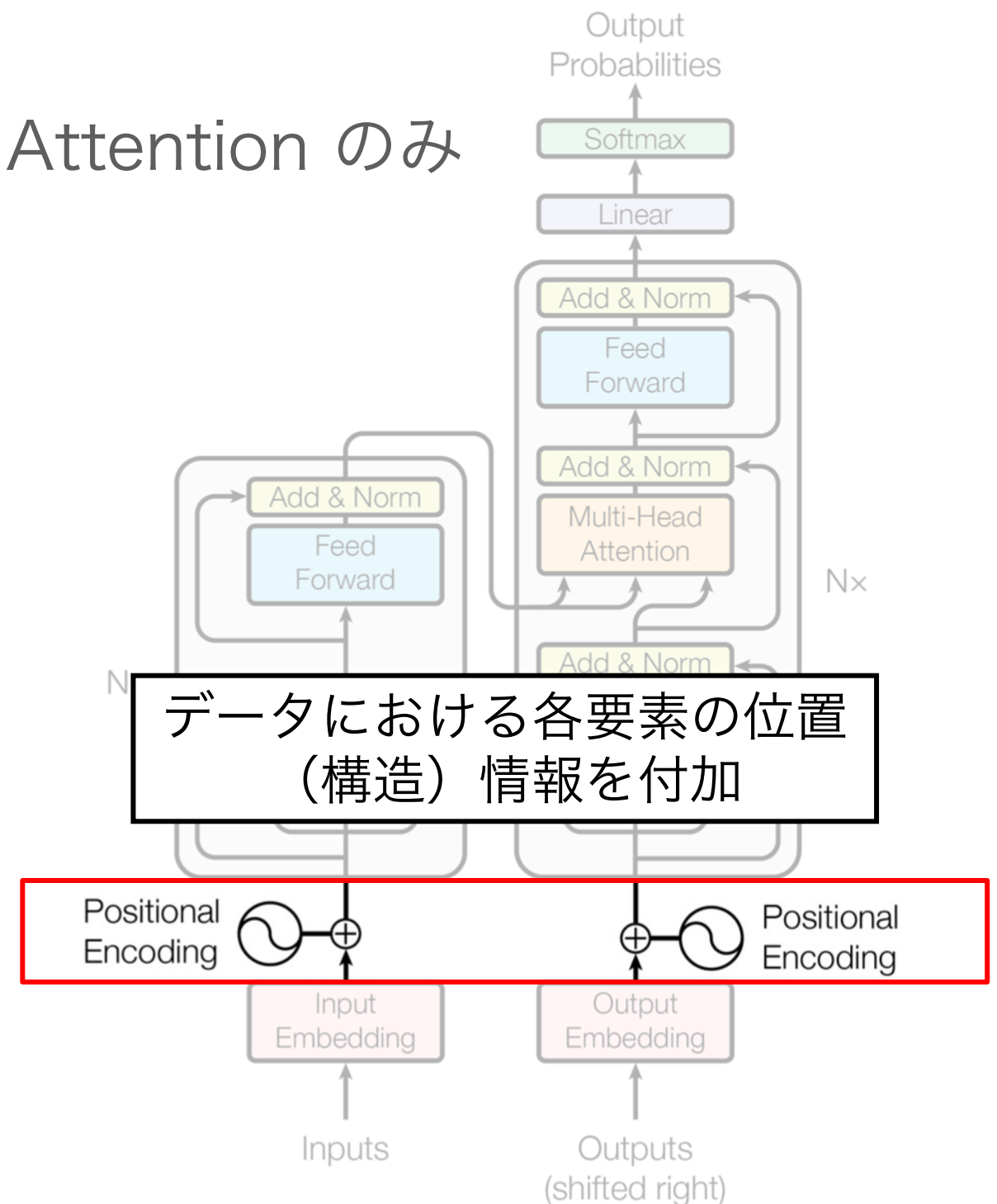
(c) 深さ優先探索

(c) AST depth-first traversal:

Assign	NameStore	SubscriptLoad	...
<empty>	elem	<empty>	...
...	NameLoad	Index	NameLoad
...	lst	<empty>	idx

構造情報の付加 (1 / 3)

- ベース
 - SRA : Sequential Relative Attention のみ
- 構造情報の付加パターン
 - Positional Encoding
 - Tree Positional Encoding
 - Tree Relative Attention
 - GGNN Sandwich



構造情報の付加 (2 / 3)

- Positional Encoding

- 三角関数を用いて入力された要素の位置を記録

$$PE_{(pos, 2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

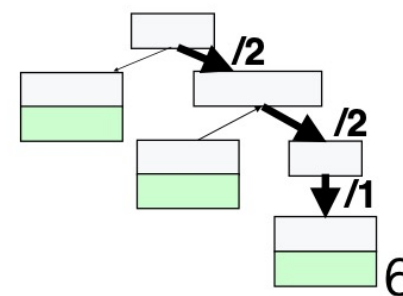
$$PE_{(pos, 2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

- Tree Positional Encoding

- ルートから各ノードへの辿る順序を記録 (バイナリ表現)
 - / : ルート, 1 : 右の子, 2 : 左の子

(d) Tree positional encodings:

1	2	3	4	5	6
/	/1	/2	/2/1	/2/2	/2/2/1



stack-like enc.

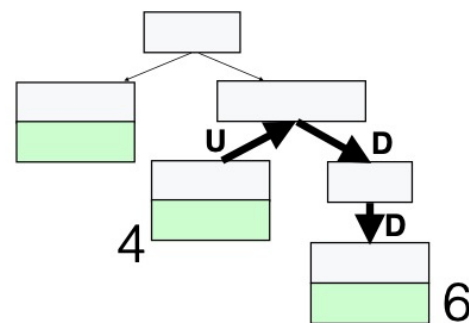
1: 000 000 000
2: 100 000 000
3: 010 000 000
4: 100 010 000
5: 010 010 000
6: 100 010 010

構造情報の付加 (3 / 3)

- Tree Relative Attention

- 各ノードから目的のノードに到達するための順序を記録
 - U : 上に移動, D : 下に移動, I : 移動なし

(e) Tree relative attention:



	1	2	3	4	5	6
1	I	D	D	DD	DD	DDD
2	U	I	UD	UDD	UDD	UDDD
3	U	UD	I	D	D	DD
4	UU	UUD	U	I	UD	UDD
5	UU	UUD	U	UD	I	D
6	UUU	UUUD	UU	UUD	U	I

- GGNN Sandwich

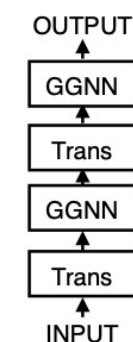
- 各ノード隣接するノードの関係を記録
 - P : 親ノード, C : 子ノード, L : 左ノード, R : 右ノード, S : 自分

(f) GGNN Sandwich:

Types of edges:

- Parent (P)
- Child (C)
- Left (L)
- Right (R)
- Self (S)

	1	2	3	4	5	6
1	S	P, L	P			
2	C, R	S	L			
3	C	R	S	P, L	P	
4			C, R	S	L	
5			C	R	S	P, L
6					C, R	S



評価実験：タスクと指標

- 評価タスク
 - VM : Variable Misuse (変数の誤った利用)
 - FN : Function Naming (関数の命名)
 - CC : Code Completion (コード補完)
- 評価指標
 - VM : accuracy
 - 全ての変数に対して2値分類
 - FN : F1
 - 予測された文字列と正解の文字列を比較
 - CC : MRR
 - 補完するコードをランキング予測

評価実験：データセット

- Python150k ^[28]
- JavaScript150k ^[30]

タスク	プロジェクト	学習データ	検証データ	評価データ
VM	Python	417K (関数)	48K (関数)	231K (関数)
	Javascript	202K (関数)	29K (関数)	108K (関数)
FN	Python	523K (関数)	56K (関数)	264K (関数)
	Javascript	186K (関数)	23K (関数)	93K (関数)
CC	Python	186K (チャンク)	20K (チャンク)	100K (チャンク)
	Javascript	270K (チャンク)	32K (チャンク)	220K (チャンク)

評価実験：内容

- 実験 1：構造情報付加による影響は？
 - コードの情報を効果的に用いるのに適している構造を評価
- 実験 2：入力データに使う情報による影響は？
 - タスクごとに重要な情報を評価

(以下省略)

- 実験 3：識別子匿名化による影響は？
 - 識別子をシンボルに置き換えたモデルの汎化性能を評価
 - プロジェクトごとに利用される識別子が異なる

実験1：結果

- GGNN SandwichはCCタスクの適用不可



全てのタスクでSRAのみのモデルは高性能

→ 新たに構造情報を付加しなくても高い性能を発揮可能

実験2：結果

AST構文+テキスト

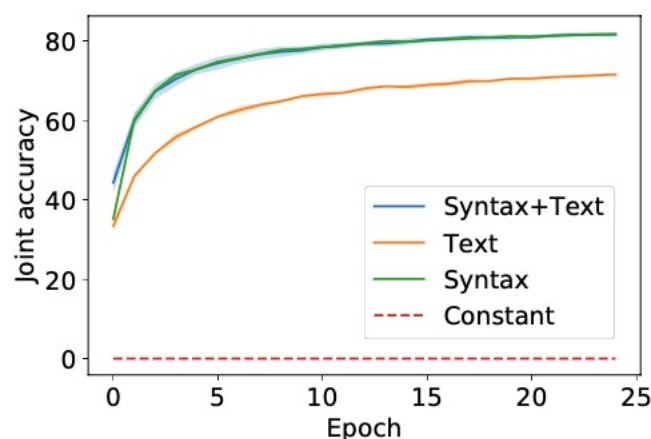
AST構文のみ

テキストのみ

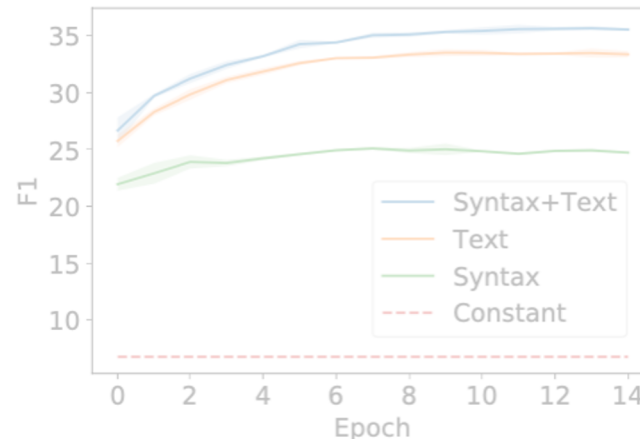
Model	Input representation
Syntax+Text	[(Assign, <empty>), (NameStore, elem), ..., (Index, <empty>), (NameLoad, idx)]
Syntax	[(Assign, <empty>), (NameStore, <var1>), ..., (Index, <empty>), (NameLoad, <var3>)]
Text	[elem, lst, idx]
Constant	Predicts the most frequent target for any input

- 入力データに使う情報の組み替え

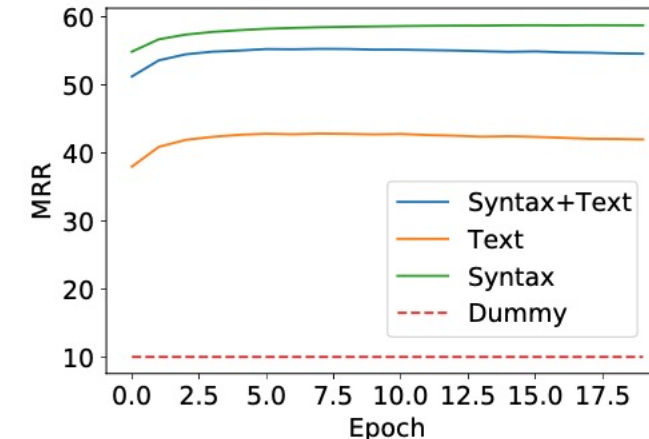
Variable Misuse – Python



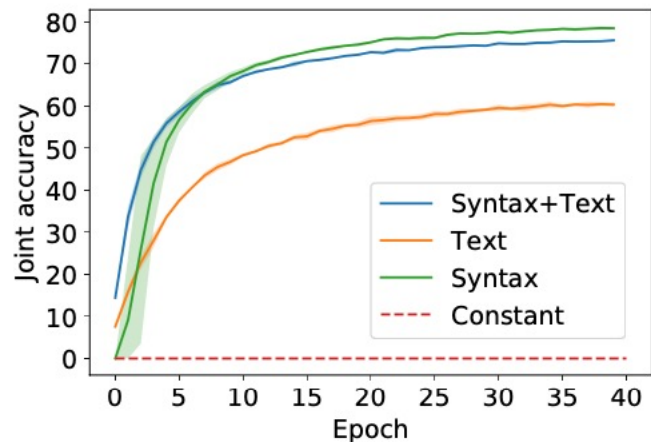
Function Naming – Python



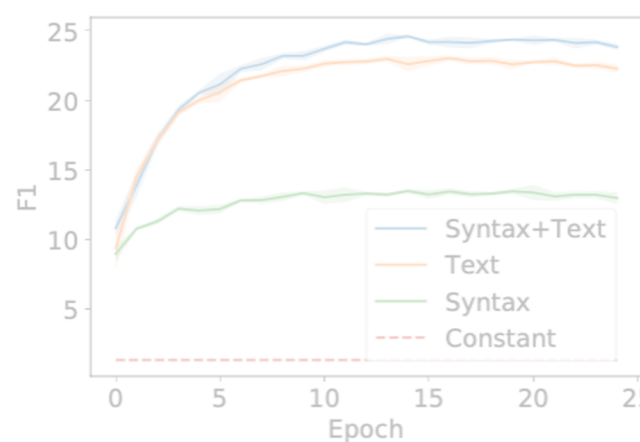
Code Completion (values) – Python



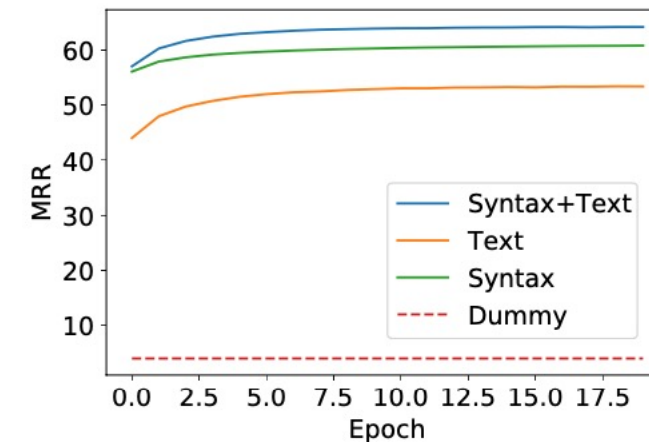
Variable Misuse – JavaScript



Function Naming – JavaScript



Code Completion (values) – JavaScript



(VMとCC) テキストのみの性能が悪い

→ 変数の誤用やコード補完はプログラムの振舞いに関係

実験2：結果

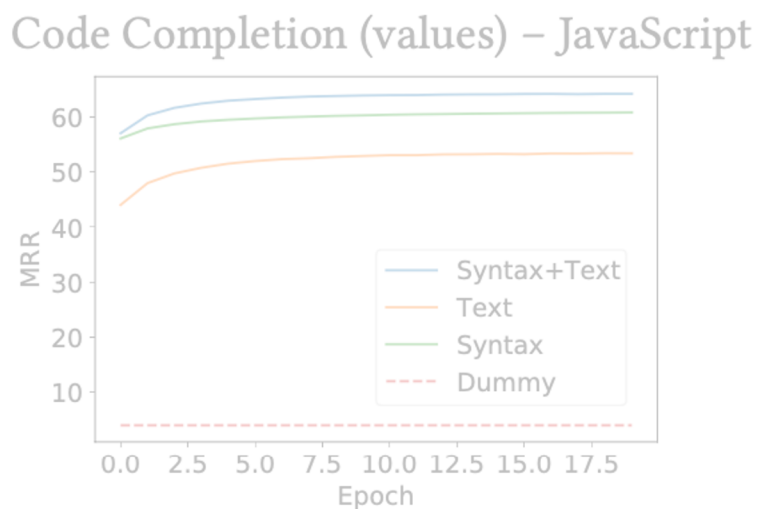
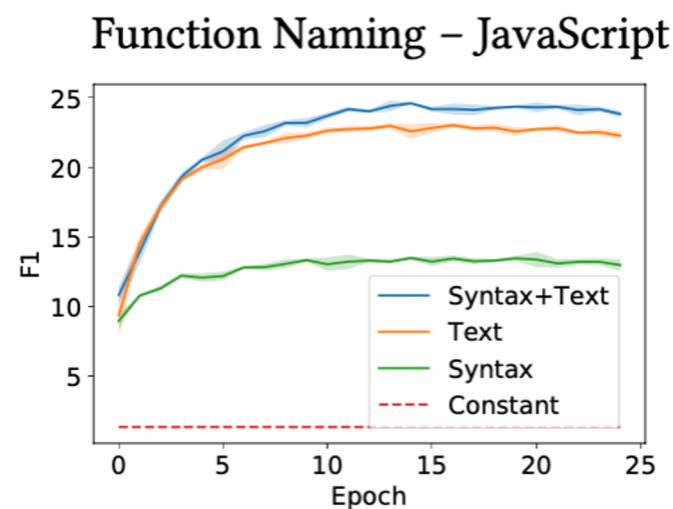
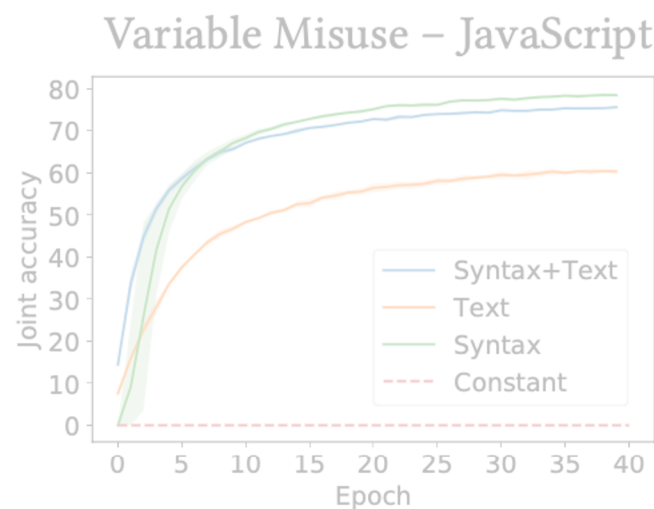
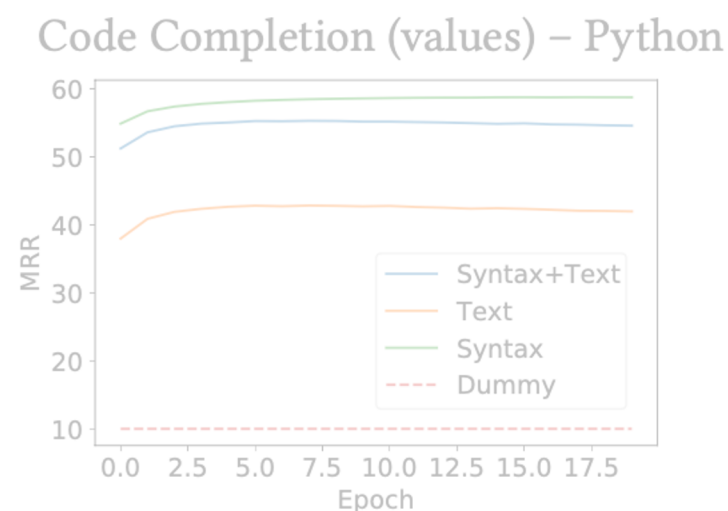
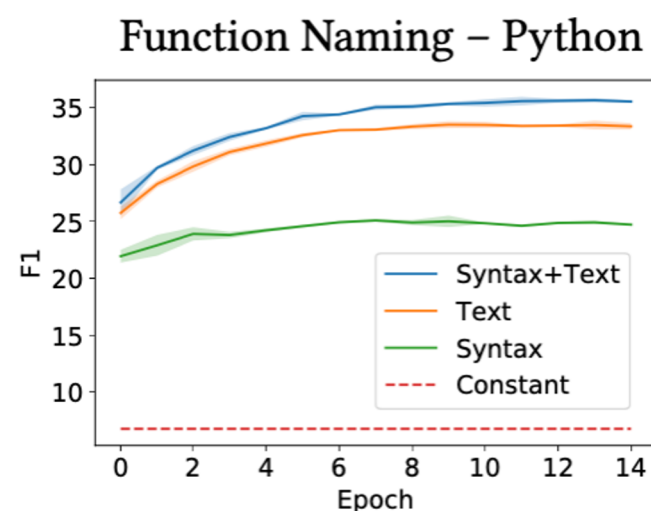
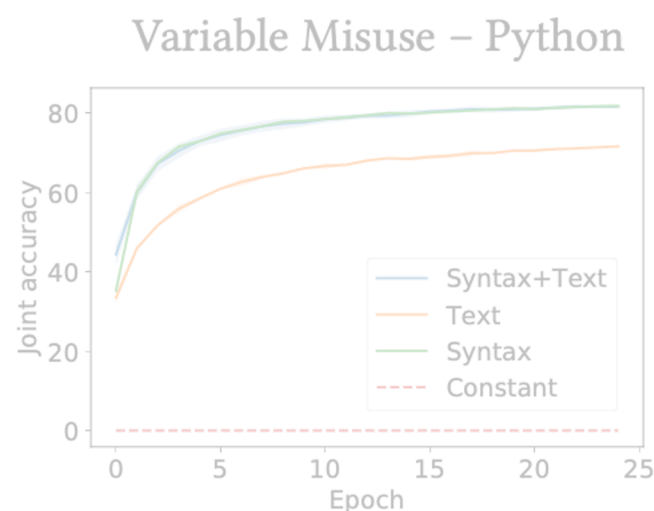
AST構文+テキスト

AST構文のみ

テキストのみ

Model	Input representation
Syntax+Text	[(Assign, <empty>), (NameStore, elem), ..., (Index, <empty>), (NameLoad, idx)]
Syntax	[(Assign, <empty>), (NameStore, <var1>), ..., (Index, <empty>), (NameLoad, <var3>)]
Text	[elem, lst, idx]
Constant	Predicts the most frequent target for any input

- 入力データに使う情報の組み替え



(FN) AST構文のみの性能が悪い

→ 関数の命名にはコードに利用されているテキストと関係

まとめ・所感

まとめ

- Transformerモデルを用いたプログラムコード処理タスクの評価
 - タスクごとに適している構造情報付加やデータ情報（AST構文やテキスト）の組み合わせ

所感

- ASTを深さ優先で整列させるだけで十分に構造が捉えられるのは興味深い
- 急に略称が出てきたり表記が揺れたりで理解するのがしんどかった