Transformerベースのコード生成モデルにおける自動テスト評価手法の提案

小原 百々雅[1,a)]　佐藤 美唯[2,b)]　梶浦 照乃[1]　富岡 真由[1]　倉光 君郎[2,c)]

概要：Transformer による言語生成モデルは，深層学習のソフトウェア工学への関心の高まりとともに，新たなコード生成の手段として注目を集めている．コード生成モデルは，原理的には，機械翻訳における出力言語を自然言語からコードに置き換えたものである．そのため，出力されるコードも，BLEU など機械翻訳の評価尺度で評価されてきた．しかし，コードは自然言語に比べ，文法規則（構文エラー）に対し敏感であり，少しの違いでも意味が全く異なる．機械翻訳の評価尺度で高スコアが得られた予測結果であっても，コードとしては正しさに欠けることが指摘されてきた．本研究では，コードの意味的な正しさをより捉えるため，ソフトウェアテストに基づく評価法，具体的には，生成した予測コードと参照コードをそれぞれ実行し，その実行結果を比較することで予測の正しさを判定する方法を試してみた．我々は，自然言語から Python コードを生成するコード生成モデルを構築し，従来の評価尺度とソフトウェアテストによる評価を比較した．本研究で得られた知見として，ソフトウェアテストによる評価は EM(Exact Match) の指標に近いことが明らかになった．

キーワード：AI によるソフトウェア開発，評価尺度，コード生成モデル，Transformer

1. はじめに

近年，機械学習の著しい発展に触発され，ソフトウェア開発やプログラミングの分野でも，機械学習を応用する技法が大きな関心 [1] を集めている．そのような大きな研究動向の中においても，Transformer[2] の登場は，深層学習によるコード生成の可能性を高める技術として，高い関心 [3], [4], [5] を集めている．

我々は，初学者へのプログラミング支援を目指し，自然言語からコードへのニューラル機械翻訳に取り組んできた [6], [7]．現在，Transformer ベースの言語生成モデルを採用することで，かなり高品質で実用的なコードが出力できる生成モデルが実現可能になってきた．しかし，伝統的な機械翻訳の評価尺度は，自然言語を対象に考案されており，少しの字句の違いで解釈が大きく変わるコードには必ずしも適していない [5], [8]．今後，よりコード生成モデルの性能を高めるためには，コード生成モデルに適した評価尺度が必要となる．

我々は，コード生成モデルの品質を評価するため，ソフトウェアテストの手法を試した．つまり，コード生成モデルが生成した予測コードとテストデータで与えられた参照（正解）コードをそれぞれ実行し，その実行結果を比較した．もし実行結果が等しければ，予測コードと参照コードは記法上の相違があっても等しいとみなすことができ，よりコードの性質に近いコード生成モデルの評価が可能になる．

本研究の目的は，実際に Transformer ベースのコード生成モデルを構築し，ソフトウェアテストによって生成されたコードの成功率を測り，その結果と従来の評価尺度の相関を調べてみることである．比較する評価尺度は，EM(完全一致), BLEU(N-gram 適合率に基づいた評価尺度)[9], ROUGE-L(最長共通部分列 (LCS) に基づく構造的類似性)[10], EditSim(レーベンシュタイン編集距離に基づく類似度)[11], Syntax(構文テストのパス率) である．

本比較実験から得られた知見は，BLEU, ROUGE-L, EditSim, Syntax は，仮に高スコアが出たとしても，コード生成モデルから生成されるコードが実行できるか明確には示せないことである．一方，EM のみは，ソフトウェアテス

[1] 日本女子大学大学院理学研究科数理・物性構造科学専攻
Graduate School of Science Division of Mathematical and Physical Sciences, Japan Women's University, 2-8-1 Mejirodai, Bunkyo-ku, Tokyo 112–8681, Japan
[2] 日本女子大学理学部数物情報科学科
Department of Mathematics, Physics, and Computer Science, Japan Women's University, 2-8-1 Mejirodai, Bunkyo-ku, Tokyo 112–8681, Japan
[a)] m1816019om@ug.jwu.ac.jp
[b)] m1916038sm@ug.jwu.ac.jp
[c)] kuramitsuk@fc.jwu.ac.jp

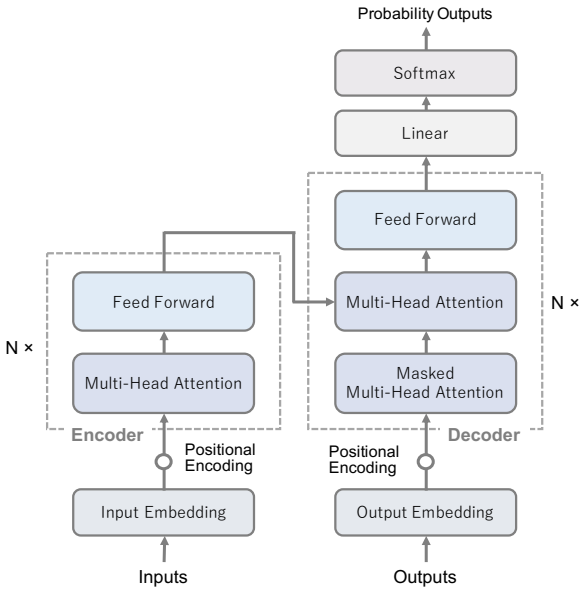

図 1 Transformer のアーキテクチャ

トによる実行パス率と近い傾向を示すことがわかった．本論文では，この比較実験の結果を踏まえ，コード生成モデルのモデル選択，モデル評価について論を深めたい．

本論文の残りの構成は以下の通りである．2 節では Transformer によるコード生成について述べる．3 節では問題定義について述べる．4 節ではコード生成モデルのためのソフトウェアテストについて述べる．5 節では評価尺度の比較実験を報告する．6 節では関連研究を概観し，7 節で本論文をまとめる．

2. Transformer によるコード生成モデル

本節では，Transformer によるコード生成モデルの原理や仕組みを概観する．また，具体例として，我々が取り組んできた自然言語記述から Python コードを生成するコード生成モデルの構築 [6], [7] を紹介する．

2.1 Transformer

Transformer は 2017 年に Vaswani らが発表した，Encoder-Decoder 型の深層学習モデルである．図 1 に示す通り，Positional Encoding により，ネットワークに再帰構造を含まない点が特徴で，GPU を用いた大量のデータによる学習に適している．ニューラル機械翻訳を含め，現在の主流ネットワークモデルとなっている．

BERT (Bidirectional Encoder Representations from Transformer) [12] は，Transformer の Encoder のアーキテクチャである．MLM(Masked Language Model) による事前学習を導入することで，多くの自然言語タスクで SOTA を達成している．

T5 (Text-to-Text Transfer Transformer)[13] は，Transformer をベースとした Encoder-Decoder 型の事前学習済

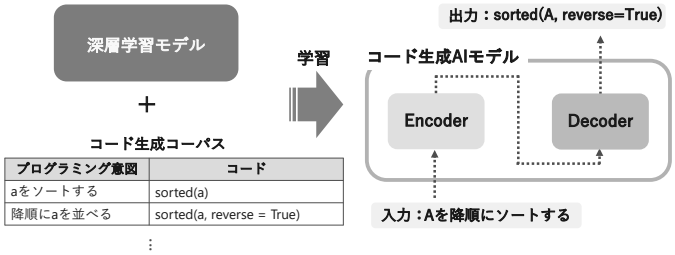

図 2 コード生成モデルの学習の流れ

みモデルである．BERT と同様に事前学習を採用し，テキスト間変換の汎用的なタスクに用いられる．

なお，Transformer の Attention 機構は，コードに不向き，つまり，変数の定義・参照間などで見られる長距離依存関係や抽象構文木の構造関係を把握できない [14] と指摘されてきた．しかし，2022 年現在，Transformer の強力なモデル能力はコード分野でも広く適用され，優れた成果をあげている．

2.2 コード生成モデル

コード生成モデルとは，ニューラル機械翻訳 [15] のコード版である．ニューラル機械翻訳では，原文とその訳文をペアとした対訳コーパスを教師データとして Encoder-Decoder 型の深層学習モデルに学習させることで翻訳モデルを構築する．コード生成モデルは，図 2 のように，自然言語の訳文の代わりに，コードを出力するように構築する．ただし，ニューラル機械翻訳の特徴として，学習させた対訳コーパスを基にある種の法則性を学び，対訳コーパスに含まれない未知の入力に対しても出力の予測が可能となる．そのため，翻訳モデルというより，生成モデルと呼ぶことが多い．

コード生成モデルは，Decoder により，コードを出力する点は共通している．出力されるコードは，コードスニペットから関数単位，ソースコードファイルまで多岐にわたる．また，コード生成モデルへの入力形式も，プログラミング課題から StackOverflow の質問など，こちらも多岐にわたる．一例を示すと，次のようなコード生成モデルがある．

- AlphaCode[16]: 競技プログラミングの問題形式からコードを生成する
- CodeX[4]: Python の docstring から関数単位のコードを生成する
- TranX+BERT[17]: 英語で記述されたプログラミングの意図からコードスニペットを生成する
- IntelliCode[5]: 途中までの入力結果からその後に続くコードを予測する
- PyNMT[6]: 日本語で記述されたプログラミング意図から式 (Expression) 単位のコードを生成する

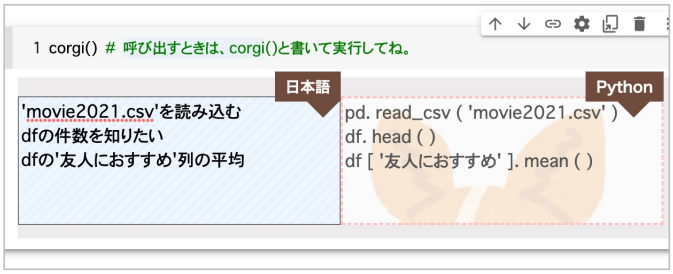

図 3 初学者向けのリアルタイムコード翻訳システム

図 4 対話型プログラミング支援システム

プログラミング意図	コード
A が偶数であるかどうか	`A % 2 == 0`
CSV ファイル A を読み込む	`pd.read_csv(A)`
カンマ区切りで A を読む	`pd.read_csv(A, sep=',')`

図 5 コーパスの例

2.3 プログラミング学習支援への応用

我々は，初学者へのプログラミング支援を最終目標として，日本語で書かれたプログラム意図から Python コードに変換するリアルタイムコード翻訳システム (図 3) を開発し [7]，最近は KOGI と名付けた対話型プログラミング支援システム?(図 4) の開発を進めている．コード生成モデル [6], ?は，このようなプログラミング支援システムのバックエンド処理系となる．

図 5 は，我々がコード生成モデルの開発に用いているコーパスの例である．プログラム意図に対応するコードは，`n % 2 == 0` のように 2 つ以上の式が複合されても構わない．(自然言語として，自然に意図を記述できる単位で記述してもよい.) 我々のコード生成モデルは，プログラミング教育の用途で用いるため，少し入力と出力に特徴がある．

- 入力は，日本語による記述を想定している．初学者のさまざまな入力表現に対応するため，直訳的な記述，意訳的な記述，あるいは省略的な記述も含める．
- 出力は，Expression 単位のコード，もしくは短いコード片とする．複数行から成る大きなコードを生成しすぎないようになっている．

我々は，所属する日本女子大学のプログラミング演習やデータサイエンス演習で，リアルタイムコード翻訳システムを用いたプログラミング学習支援を行っている．コード翻訳の評価は，コード生成モデルの入出力結果をサンプル抽出し，人手による正解/不正解の判定に頼っている．講義に安定して活用できる正解率は，80%〜90%程度であるが，モデル構築に失敗すると，40%強の正解率に落ちることもある．安定的にシステムを運用するためには，自動的な評価尺度が求められる理由である．

表 1 コード生成モデルの出力例

ケース (1)	完全一致
(1-1)	**ファイルの拡張子なしのファイル名を求める**
	(予測) `os.path.splitext(os.path.basename(file))[0]`
	(参照) `os.path.splitext(os.path.basename(file))[0]`
(1-2)	**リストの最初の n 要素を捨てる**
	(予測) `aList[ n : ]`
	(参照) `alist[n:]`
ケース (2)	**部分一致**
(2-1)	**データフレームを並べ直す**
	(予測) `df.sort_values(' キーとなる列').reset_index()`
	(参照) `df.sort_values(' キーとなる列')`
(2-2)	**双方向キューの要素を左に n 個分ローテンションする**
	(予測) `deq.rotate(n)`
	(参照) `deq.rotate(-n)`
ケース (3)	一致しないが意味が等しい
(3-1)	**コサイン x を求める**
	(予測) `sin(x)`
	(参照) `math.sin(x)`
(3-2)	**中央値で配列をビン分割する**
	(予測) `pd.qcut(ds, 2)`
	(参照) `pd.qcut(aArray, 2)`
ケース (4)	不一致
(4-1)	**外れ値にロバストな標準化を行う**
	(予測) `sklearn.linear_model.HuberClassifier()`
	(参照) `sklearn.preprocessing.RobustScaler()` `.fit_transform(データ)`
(4-2)	**データフレームを行によって降順で並べる**
	(予測) `df.merge(columns={column::::::: dropna')`
	(参照) `df.sort_index(ascending=False)`
ケース (5)	そもそも正解がない
(5-1)	**バイトの時間を 1 時間間違えた**
	(予測) `int(time.time() + 7) // 8`
	(参照)

3. コード生成モデルの品質

Transformer によるコード生成モデルは，深層学習を応用したソフトウェア開発・コーディング支援において，今後，重要な構成要素になると期待される．しかし，現状では，コード生成モデルが出力するコードの品質を評価する方法が定まっていない．本節では，コード生成モデルが出力するコードを見ながら，既存の評価手法を概観する．

3.1 生成されたコードの分析

まず，問題を理解するためコード生成モデルが生成した実際のコードを確認するところから始めたい．

表 1 は，過去 KOGI プロジェクトで開発したコード生成モデルが出力した実際のコード例[*1]を示している．入力例

[*1] KOGI プロジェクトでは，コーパスの品質を向上させながら，

は，(1-1) から (5-1) のようにラベル付けされている．コード生成モデルは，このような入力例からコードを予測する．(予測) ラベルは予測されたコード，(参照) ラベルは検証データに含まれる正解コードをそれぞれ示している．

我々は，ここでは，コード生成モデルが出力する予測コードを大きく次の 5 つのケースに分類している．

(1) 完全一致
(2) 部分一致
(3) 一致しないが意味が等しい
(4) 不一致
(5) そもそも正解がない

完全一致は，予測コードと参照コードが完全に一致するケースである．予測コードが，(1-2) のように空白など参照コードと異なるコードレイアウトを持つことがあるが，これらも完全一致の一種と見なす．

部分一致は，コード生成における主要なコード片が含まれており，人間から見て惜しいと感じられるコードが出力される例である．ただし，利用者の立場からいうと，参照コードに近い分，人間が過ちに気づきにくく，少しの間違いによって全く異なる結果，もしそのまま実行したら望ましくない結果を招く恐れがある．

コード生成モデルは，参照コードとは異なるが，意味的に同じ，もしくは自然言語記述から解釈しても正解と見なして構わないコードを生成することがある．このようなケースが (3-1), (3-2) である．このような意味的に同じコードが出力されるケースは，Transformer モデルの生成能力，もしくは訓練データに依存すると考えられる．

ほぼランダムは，コード生成の完全な失敗ケースといえる．(4-1) のようにプログラミング言語の構文を維持している場合と (4-2) のように構文的にも間違っている場合がある．どちらにしても，利用者はコード生成モデルから出力された結果からほとんど有意義なヒントを得ることはない．

最後のケース (5) は，特殊なケースである．利用者は，自然言語であらゆる記述を入力することができ，コード生成モデルはどのような入力であっても何らかのコードを出力する．我々は，コードに変換できない入力は，コードに変換しないのが正解と考えている．

我々はコード生成モデルの利用者として，KOGI のようなプログラミング支援を想定している．このような場合，ケース (1) やケース (3) は望ましいコード生成の結果といえる．一方，ケース (2) は利用者が間違いに気づかないと正しくないコードをそのまま利用する心配もある．なお，コード生成モデルの想定するアプリケーションが異なれば，コード生成モデルの評価基準が異なることに留意されたい．

コード生成モデルをアップデートしている．そのため，現在のコード生成モデルで同じ出力が得られることは保証されない．

3.2 字句に基づく評価

字句に基づく評価尺度は，機械翻訳の評価に広く使われている．記述の厳密さを要するプログラミング言語を扱うコード生成モデルの評価においても，一つの指標となり得る．

3.2.1 BLEU

BLEU は，N-gram 適合率に基づいた評価尺度であり，1-gram から N-gram についての適合率の幾何平均から評価値が算出される [9]．評価値は 0〜1 で表現され，予測文と参照文が完全に一致する場合に 1 となる．

3.2.2 ROUGE-L

ROUGE-L は，LCS に基づき構造的類似性を考慮した評価を行う評価尺度である [10]．LCS とは，予測文と参照文で共通の連続する N-gram のうち最長のシーケンスである．ROUGE-L では，参照文の単語数に対する LCS の長さの割合を再現率，予測文の単語数に対する LCS の長さの割合を適合率とし，再現率と適合率の F 値を評価値として算出する．機械翻訳だけでなく，自動要約においても一般に使われる．

3.2.3 レーベンシュタイン類似度

レーベンシュタイン類似度は，レーベンシュタイン距離を正規化し，1 から引くことで文字列間の類似度を測る指標である．レーベンシュタイン距離とは，予測文から参照文に変形するとき，何文字の編集（挿入，置換，削除）が必要かを測定したものである [11]．レーベンシュタイン類似度は，コード補完などのタスクにおける評価尺度としても利用されている [5]．

3.3 構文ベースの評価

自然言語は，ノイズと呼ばれる字句の些細な相違に対してある程度寛容的であるため，字句に基づく評価尺度は重宝される評価指標である．一方，コードはわずかな字句の差異でも決定的な動作不良になる．コード生成モデルによる予測コードは，以下のように，ノイズ的な字句が混在するケースが少なくない．

```
print("hello",,, "world")
```

ノイズは，仮に BLEU や ROUGE-L が好評価であっても，ユーザ視点から見ると，明らかな間違いと認識される．したがって，ノイズの少なさや構文的な正しさは，コード予測モデルにおいて重要な評価基準になる．

構文に基づく評価は，プログラミング言語の構文解析器を通すことで，判定することができる．しかし，予測コードが構文解析器を無事にパスして，構文的な正しさが評価できただけで，参照コードに対する近さは何もわからない．

3.4 意味的な等しさの評価に向けて

プログラミング言語では，コードの書き方は異なるが意味は同じというコードは無数にある．コード生成モデルでは，このような意味的に同じというコードが生成されることがある．

ここでは，コード生成モデルが，参照コードに対して，次のような予測コードを生成した場合を考える．

（参照） `s[:len(x)] == x`
（予測） `s.startswith(x)`

これらの２つのコードは，共にどちらも文字列 s の先頭が x に等しいかどうか判定するコードである．意味的にはほぼ同じコードといえるが，字句ベースや構文ベースの評価では意味まで含めた判定はできない．ただし，コード生成モデルの評価としては，これらの二つのコードが等しいものと評価することは望ましい．

プログラミング意味論では，コードが意味的に等しいかどうかを判定するため，公理的意味論，表示的意味論，操作的意味論などの形式的性質から議論してきた．コードの意味の等しさを機械的に判定する汎用的な方法は存在していない．

4. ソフトウェアテストによる評価

我々は，ソフトウェアテストの手法を導入することで，コードの意味的な正しさに踏み込んだコード生成モデルの評価を試みた．

4.1 ベースアイディア

我々のアプローチは，コード生成モデルが予測したコードを実行し，その実行結果を参照コードの実行結果と比較することで，コードの意味的な正しさを評価することである．これは，操作的意味論の「式の評価」に相当する．

まず，アイディアを示すため，(参照) `s[:len(x)] == x` に対し，２種類の予測コードが生成された場合を考える．ここで，テストデータを以下の通りに与えた環境で評価すると評価値が得られる．

（テストデータ）	`s = "aba"` `x = "a"`	
（参照コード）	`s[:len(x)] == x`	True
（予測コード a）	`s.startswith(x)`	True
（予測コード b）	`s.endswith(x)`	True

もし両者の評価結果が異なれば，参照コードと予測コードは明らかに異なると判断できる．一方，評価結果が等しくても意味的に等しいとは限らない．ここで，(予測コード b) の `s.endswith(x)` は，文字列 s の末尾が x に等しいかどうか判定し，明らかに参照コードと異なるが，テストデータのケースでは，どちらも等しくなる．

ソフトウェアテストでは，通常，テストデータを丁寧に開発することで，偶然にテスト結果が一致するリスクを提言している．しかし，コード生成モデルの開発においては，数万件の教師データに対し，テストデータを開発するコストが見合うものではない．

我々は，テストデータをランダムに生成し，複数回実行することで，確率的に近いかどうか判定することにする．

4.2 テストデータの収集

テストデータは，Python ファイルのグローバル変数の定義文から収集する．ここでは，次のようなグローバル変数の場合を考える．

```
adict = {'A': 1, 'B': 0}
```

我々は，Python のビルトイン関数 `exec()` を用いて，グローバル変数の定義を実行し，ローカル変数の環境マッピング `local_vars` から変数名に対応する値を得る．

```
exec('adict = {'A': 1, 'B': 0}',
    None, local_vars)
```

なお，複数の Python ファイルからテストデータを回収するため，同名で異なる値が定義されることも多い．テストデータの収集としては，変数名に対して複数の値をリストとして保持する．このとき，値の型が統一されてなくても構わない．

4.3 ランダムテスト

参照コードと予測コードのテストは，テストデータの抽出と同じく，ビルトイン関数 `exec()` を用いて実行する．こちらは，実行前と実行後のローカル環境の変化を識別するため，階層化されたチェーンマッピングを用いて，環境を実装する．

具体的なテストの手続きは次の通りになる．

(1) 参照コードから変数名を抽出する．
(2) （前節で抽出した）テストデータのリストから変数名が一致する値をランダムに選ぶ．
(3) ミュータブルな値は，`copy` モジュールで複製しておく．
(4) まず，参照コードに対して，Python の `exec()` で実行する．
(5) もし (3) で型エラーなどのランタイムエラーが発生したら，テストデータの乱択 (2) に戻る．
(6) もしエラーが発生しなければ，ローカル環境の中から変数値が変化した値を抽出する．
(7) 予測コードも，(3) で複製されたテストデータを用いて実行し，変化した値を抽出する．
(8) 参照コードと予測コードの実行結果を比較する．

```
class Missing:
  def __init__(self, name):
    self.msg = name

  def __str__(self):
    return self.msg

  def __repr__(self):
    return self.msg

  def __getattr__(self, name):
    return Missing(f'{self.msg}.{name}')

  def __call__(self, *args, **kwargs):
    if len(kwargs) == 0:
    return Missing(f'{self.msg}{args}')
    else:
    return Missing(f'{self.msg}{args}{kwargs}')
```

図 6 Missing Method の実装 (抜粋)

テストツールは，ひとつの参照コードと予測コードの対に対して，以上の手順を複数回，繰り返す．Python では型に起因するエラーは `TypeError` や `ValueError` のようなランタイムエラーとして報告されるため，これらが含まれるテストは無視し，ランタイムエラーの発生しないテスト結果のみ評価する．複数回繰り返し，一度もテスト結果が評価できなかった場合は，untested とする．

なお，`exec()` はコード（ステートメント）の実行に用いる関数で，ビルトイン関数 `eval()` ように評価値は得られない．そのため，構文からステートメントか式かを判定し，式の場合は代入形式に変換してから実行する．

4.4 Missing Method

コード生成モデルの出力には，以下のように実行しにくいコードが含まれる．

- `sys.exit(0)`：プログラムを強制終了する
- `random.random()`：乱数を生成する（毎回，実行結果が異なる）
- `plot.scatter(X, Y)`：Matplotlib で散布図を描画する

我々は，このようなコードをテストするため，Missing Method を用いる．

Missing Method とは，メソッドコール時にメソッドが存在しなかったときの処理を定義したメソッドである．Python ではネイティブな言語機能としてサポートされていないが，我々は，演算子オーバーロードの機能を用いて，同等の機能を実装した．図 6 は，Missing Method の実装例（抜粋）である．

次のようなコードをテストしたい場合を例として考える．

```
io.open('file.txt', mode='w')
```

本当のメソッドを呼び出す代わりに Missing Method を用いたオブジェクトに置き換える．

```
io = Missing('io')
```

すると，Missing クラスの`__getattr__`でメソッド名を取得し，`__call__`で引数の値を取得し，最終的な実行結果として，`io.open('file.txt', {mode: 'w'})` のような文字列を返すようになり，実行結果を比較できるようになる．

Missing Method は，書き方が異なる同じコードを判定できなくなるが，異なるコードを実行したとき，同じ結果になることはない．

5. 比較実験

本節では，実際に Transformer によるコード生成モデルを構築し，生成されたコードの品質を測定する．

5.1 評価尺度

本実験で用いたコード品質の評価尺度と算出方法は，次の通りである．

- **Exact Match** (EM): 予測コードと参照コードの字句が完全に一致した割合．空白などのコードレイアウトの影響を取り除くため，コード整形器 black を通して測定している．
- **BLEU**: 機械翻訳の標準的な自動評価尺度．自然言語ツールキット NLTK の `sentence_bleu` を用いて，Python 標準 Tokenizer を用いて字句分割し，最大 4-gram の BLEU-4 として算出している．
- **ROUGE-L**: 文書要約の代表的な自動評価尺度．語順の並びに依存しない．評価用フレームワーク SumEval の `RougeCalculator` を用いて算出している．
- **EditSim**: レーベンシュタイン編集に基づく類似度．Levenshtein モジュールで算出している．
- **Syntax**: Python の標準的な構文解析器に通し，構文エラーが発生せずに構文解析に成功したコードの割合を求める．
- **Run Match** (RunM): 4 節で述べたソフトウェアテストによりコードを実行し，実行結果が一致した割合を求める．

5.2 実験モデルの構築

我々は，コード品質の測定を実施するにあたり，故意に学習回数 (epoch) を減らしたコード生成モデルを構築し，

表 2 コード品質の測定結果

#	EM	BLEU	ROUGE	EditSim	Syntax	RunM
1	0.00	31.57	46.41	52.77	76.25	0.00
2	0.00	45.31	52.88	61.29	89.22	0.00
3	0.79	52.24	59.33	67.33	88.22	0.79
4	1.40	55.19	63.64	70.42	94.61	1.40
5	2.60	56.98	67.59	73.77	93.41	2.40
6	7.98	60.72	70.18	76.80	99.40	7.89
7	14.17	62.66	71.53	78.52	99.00	**14.57**
8	31.33	68.43	76.64	81.58	99.40	**31.93**
9	46.11	71.54	80.45	85.28	99.20	**46.50**
10	60.08	75.53	83.33	87.52	99.80	59.48
15	76.84	84.18	89.94	92.49	99.40	74.25
20	82.23	86.56	92.51	94.41	99.20	78.44
25	87.62	90.17	95.60	96.77	99.80	82.24
30	89.62	91.40	96.18	97.29	99.80	83.03
正解	100.00	97.69	99.85	100.00	100.00	96.93

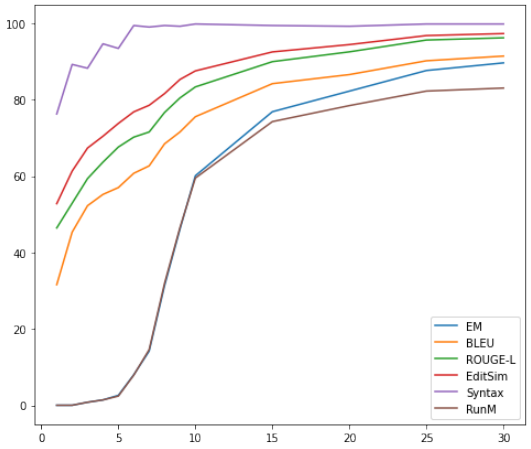

図 7 学習回数と測定された評価尺度の推移

コード品質のばらつきを人為的に産み出して，各評価尺度での測定を行った．

構築したモデルは，標準的な構成 [2] で定義された Transformer を用いた．事前学習は行っていない．ハイパーパラメータは，Encoder と Decoder は 6 層でエンベッディングの 512 次元，学習率は 0.0003，ミニバッチ学習のサイズは 32 であった．

学習には，KOGI プロジェクトで機械生成した日本語-Python の対訳コーパスを用いた．訓練データは 11367 件，検証データ 2638 件であった．なお，学習回数とコーパスは，注意が必要である．我々は，独自のデータ水増し (Data Augmentation) を加えたコーパスを用い，最終的に実行可能なコードが出力できるように工夫している．そのため，学習回数が少なくてもかなり高いスコアを示す評価尺度もある．ただし，コーパスが異なれば，学習回数と各評価尺度の示す結果は異なる．

表 2 は，学習回数 (epoch) ごとの EM, BLUE, ROUGE-L, EditSim, SynM, RunM の測定結果を示している．測定値は，0〜100 のスケールに統一し，100 に近いほど良い結果と解釈される．最終行には，予測コードの代わりに正解コード対して算出した各評価尺度の実測最高値を記載してある．また，学習回数を横軸，表 2 の測定結果を縦軸として変遷を図示し，傾向を確認した（図 7）．

RunM は，生成されたコードを実行した結果，正解と同じ結果が得られるか示しており，より人間の感覚に近い正確さを反映している．BLEU, ROUGE-L, EditSim は，既存の評価尺度ではメジャーな指標であるが，RunM との相関は弱い．特に，BLEU が 60 を超えても，RunM は 10% 程度に止まり，実行可能な意味での正しいコードを生成できていないことが読み取れた．BLEU が 70 を超えると，RunM は 50%程度に跳ね上がる．この BLEU の評価尺度の 1/10 の区間に，コードの品質が大きく変わっている．

BLEU スコアは，コーパスに強く依存するため，BLEU と RunM の関係を一般化できないと考えられる．したがって，BLEU は生成されたコードの正確さ，言い換えると，どの程度の BLEU スコアなら生成されたコードが実行可能かを判断することにはほとんど役に立たないと結論づけることができる．

Exact Match と RunM は，興味深いことに，ほとんど同じ傾向を示した．これは，Exact Match したコードは実行結果が等しいため，ある意味，妥当過ぎる結果である．この結果が意味することは，「コードの書き方は異なるが等しい意味のコード」はほとんど生成されていない事実を示している．RunM 列の太字で示した部分は，Exact Match より高くなった結果であるが，モデルがデータに適合しきっていない段階の現象である．（検証データの loss 値から分析する限り，過学習にはなっていない．）ソフトウェアテストでは，（少なくとも我々の 4 節の実装では）実行結果を比較できないコードも含まれるため，RunM は Exact Match より低い値になる．

Syntax は，今回の実験では常時，高い値を示した．過去，コード生成モデルでは，構文レベルで正しくないコードが議論の対象になることもあったが，現在，Transformer などの優れたネットワークモデルが登場し，適切にモデルを構築すれば，構文レベルの間違いはあまり悩まなくて済むといえる．

5.3 ソフトウェアテストの展望

続いて，ソフトウェアテスト（RunM）を分析していきたい．表 2 の正解行が示す通り，RunM は同じコードを比較しても 1.0 にならない．これは，`random.random()` のような実行結果が一致しないコードが含まれ，どうしても不正解になってしまうためである．

もう少し，テストツールの判定結果の内訳に着目したい．今回の実装では，テストデータをランダムに選んで 10 回試行し，実行エラーを発生することなく，4 回以上実行結果が等しい場合のみ，テストにパスしたと判定している．RunM は，このパスした割合である．一方，一回でも実行

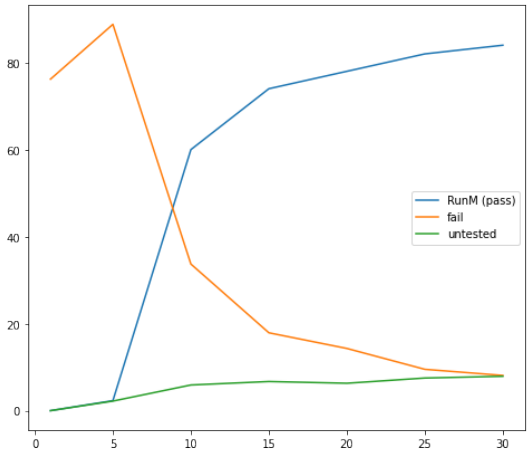

図 8 RunM のパス，失敗，未テストの推移

結果が一致しなければ，失敗 (fail)，10 回試行して，失敗しなかったケースは，未テスト (untested) になる．図 8 は，学習回数に対する RunM のパス，失敗，未テストの推移を示している．untested は，10%未満で推移したことから，未テストケースだけコード生成の品質が特に変わることもないため，有効なテストが行えたといえる．

一方，Missing Method の利用率が 76.8%と高いことがわかった．これは，テストツールを停止させないため，`sys`, `os` などのモジュールを強制的に Missing Method を使うように指定したことも一因である．Missing Method の利用は，異なるコードを正解として誤判定することはないが，「書き方が異なるコードを同じ」と判定することも難しくなり，取りこぼしている可能性はある．

5.4 議論と応用

ここで，総合的な視点で結論を論じたい．今回，ソフトウェアテストを導入したコード生成モデルの評価を試み，RunM として測定し，Exact Match とほぼ同等の傾向を示すことがわかった．一方，テストツールの開発コスト，Missing Method による判定能力の制限なども勘案すると，我々は，Exact Match の評価尺度で代替できると結論に至った．

ただし，本結論の採用には 2 点の注意が必要である．Transformer とコード生成モデルへの応用は，新しい試みでまだわかっていないことも多い．今後，自然言語処理の応用例と同じく，より生成能力の高いモデルが構築できるようになれば，書き方は異なるが意味が同じコードがより頻繁に生成されるようになるかも知れない．そのような場合は，RunM のようなソフトウェアテストを用いるのは有用かつ必要となる．

もうひとつは，Exact Match による過学習の危険性である．最初の注意点と似ているが，Exact Match の高さを追求したモデルを構築すると，訓練データにより適合し，モデルの持つ生成能力が低下する恐れがある．今回も，表 2 の太字で示したとおり，わずかであるがそのような傾向が見られた．

表 3 KOGI コーパスのモデル評価

KOGI	EM	BLEU	ROUGE-L	EditSim
Transformer	32.04	71.41	80.17	77.50
Megagon/T5	47.99	74.72	83.96	87.93
Google/mT5-small	85.18	90.36	94.74	99.47

このような点に留意すれば，Exact Match を用いてモデルを議論するのは有効である．我々の経験に基づいて 2 点ほど，Exact Match によるモデル評価の実例を報告したい．

5.4.1 事例: KOGI プロジェクト

ひとつは，KOGI プロジェクトのモデル構築のケースである．過去 KOGI プロジェクトでは，以下のネットワークモデルを用いて，コード生成モデルの構築を行い，ユーザにプログラミング支援を提供してきた．

- **Transformer**: 事前学習なしの Transformer
- **MegagonLabs/T5**: Megagon Labs 社が Hugging Face から公開している日本語事前学習済み言語モデル
- **Google/mT5-small**: Google 社が Hugging Face から公開している多言語事前学習済み言語モデル

表 3 は, KOGI コーパスのモデル評価の例である．BLEU, ROUGE-L, EditSim などに注目すると，Transformer も Megagon/T5 も十分なコード品質を達成している印象を受け，実際 EM が高いのは過学習の疑いを持っていた．そして，コード生成モデルをアプリケーションに組み込むときは，レスポンスタイムなどの他の要因も勘案することになる．その結果，KOGI プロジェクトでは，よりレスポンスタイムの短い Transformer や Megagon/T5 を採用する時期もあった．

今回の実験結果より，Exact Match は，コードを実行したときの品質に読み替えられることができ，少なくとも表 3 の場合では，mT5-small が明らかな候補となる．その結果，KOGI のユーザ体験 [7] が大幅に向上した．

5.4.2 事例: CoNaLa チャレンジ

もうひとつの実例は，CoNaLa チャレンジ [18] のケースである．CoNaLa コーパス [19] は，米 CMU が中心となって開発した英文/Python コードの対コーパスである．CoNaLa 公式ベースラインモデルは BLEU10.58, 2022 年現在の SOTA モデル (TranX+BERT[17]) は BLEU34.20 が報告されている．表 4 は，研究会報告 [18] における我々の分析結果である．最新の CodeT5 モデルでは，CoNaLa コーパスでも SOTA に近づいた．EM がほとんど 0.0 に近いことが示すとおり，我々の実験したモデルが生成したコードは実行に耐えうる品質ではなかった．

一般に，機械翻訳において BLEU スコアが高過ぎるのは望ましいことでない．CoNaLa コーパスは，BLEU スコアが高くなりすぎないようにコーパスが開発され，多くの

表 4 CoNaLa コーパスのモデル評価

CoNaLa	EM	BLEU	ROUGE-L	EditSim
Transformer	0.00	8.74	9.23	27.46
mT5-small	0.00	18.03	27.51	43.67
mT5+Python	0.00	18.21	29.32	43.94
CodeT5 -small	0.04	33.73	49.02	58.01
mT5-base	0.01	24.64	37.08	49.69
CodeT5 -base	0.04	31.96	46.21	56.41

コード生成モデルの開発に貢献してきた．一方，EM に着目すると，どのモデルも実行可能なコードというレベルでは，ほとんど差のない結果といえる．

6. 関連研究

近年，Transformer をベースとした深層学習モデルの発展は著しく，コード生成タスクとして研究も盛んに行われている [3], [20]．同時に，生成されたコードを適切に評価する尺度は重要になるが，機械学習や文字列マッチなどの旧来の尺度に頼ったものが多い．

BLEU は，機械翻訳の自動評価尺度として開発されたものであるが，コード生成モデルでも主要な評価尺度として現在も広く採用されている．しかし，BLEU は字句を考慮した指標であるため，構文的な正しさや論理的な正しさを無視した評価となり，SMT(Statistical Machine Translation) ベースのコード生成の時代からコード生成モデルの評価には相応しくないという指摘 [21], [22] がなされてきた．本研究の主張も，これらの主張に反するものではなく，Transformer が登場してコード生成モデルの能力が大きく向上した中，コードの実行結果と比較して，BLEU の限界を実験的に示したことが新しい．

ソフトウェアテストによるコード生成モデルの評価は，計算に基づく正解率 (Computational Accuracy) と呼ばれる．論文 [23] では，評価に用いるデータセットとなる関数それぞれにユニットテストを用意し，生成されたコードがユニットテストをパスするかどうかで評価している．同様に，CodeX[4] では，Human Eval が採用されている．これは，164 個のプログラミング問題からなるデータセットであり，このデータセットを用いたユニットテストをモデルの評価として取り入れている．また，AlphaCode[16] では，競技プログラミングのコンテスト Codeforces における順位でモデルを評価している．ただし，どの場合もデータセットに依存した評価方法で，普遍的な評価尺度とはいえない．

CodeBLEU[8] は，コード生成モデル用に改良された BLEU である．字句 (token), キーワード (keyword), 構文 (syntax), データフロー (dataflow) の 4 つの点から総合的に評価しオリジナルの BLEU のように表面的な一致を考慮するだけでなく，文法的な正しさや論理的な正しさも考慮することができる．

本研究では，Exact Match がコード実行の結果と高い相関性が示された．ただし，Exact Match は，厳し過ぎる上，意味的に同一なコードを評価できない分，モデルを過小評価してしまう傾向 [8] があると指摘されている．

7. むすびに

Transformer は，自然言語処理分野で優れた言語生成能力を示し，同時にコード生成モデルへの応用も強く期待されている．生成されたコードは，自然言語とは異なり，ノイズに弱く，少しの違いでも意味が大きく異なることも少なくない．従来，BLEU などの機械学習向けの評価尺度でコード生成モデルも評価されることが多かったが，コードに適した新たな評価尺度の提案が期待されている．

本研究では，ソフトウェアテストに基づく評価方法に取り組んだ．これは，生成されたコードを実際にプログラムとして実行し，その実行結果を正解コードの実行結果としてマッチする方法である．単なる字句レベルの類似を超えたコードの意味的な正しさに踏み込んだ評価が可能となる．

我々は，提案のソフトウェアテストに基づく評価法をランダムテストの形で実装し，実際に構築したモデルで比較実験を行った．実験結果は，BLEU, ROUGE-L, EditSym などの従来の評価尺度は，どれも同じ傾向を示したが，コードが実行できるかどうかの正しさとは大きくかけ離れた指標であることが示された．一方，ソフトウェアテストによる評価も，Exact Match の指標と類似した結果となった．本研究の結論は，ソフトウェアテストツールの開発コスト，Transformer ベースのモデルのコード生成能力を勘案し，「Exact Match で十分にコード生成モデルの実用性は評価できる」である．ただし，この結論は，ソフトウェアテストが無駄や必要性がないわけでなく，コードの正しさや信頼性を評価する手段として確立されたソフトウェアテストに裏付けられたものであると付記しておきたい．

Transformer ベースのコード生成モデルは新しく，その生成能力は未知な部分が少なくない．今後は，Transformer のコード生成能力の理解を深め，コードの意味的な正しさに踏み込んだ評価方法の更なる検討を行っていきたい．

参考文献

[1] Allamanis, M., Barr, E. T., Devanbu, P. and Sutton, C.: A Survey of Machine Learning for Big Code and Naturalness, *ACM Comput. Surv.*, Vol. 51, No. 4 (online), DOI: 10.1145/3212695 (2018).

[2] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. and Polosukhin, I.: Attention is All You Need, *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, Red Hook, NY, USA, Curran Associates Inc., p. 6000–6010 (2017).

[3] Wang, Y., Wang, W., Joty, S. and Hoi, S. C. H.: CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation

(2021).

[4] Chen, M. et al.: Evaluating Large Language Models Trained on Code (2021).

[5] Svyatkovskiy, A., Deng, S. K., Fu, S. and Sundaresan, N.: *IntelliCode Compose: Code Generation Using Transformer*, p. 1433–1443 (online), available from ⟨https://doi.org/10.1145/3368089.3417058⟩, Association for Computing Machinery (2020).

[6] Akinobu, Y., Obara, M., Kajiura, T., Takano, S., Tamura, M., Tomioka, M. and Kuramitsu, K.: Is Neural Machine Translation Approach Accurate Enough for Coding Assistance?, *Proceedings of the 1st ACM SIGPLAN International Workshop on Beyond Code: No Code*, BCNC 2021, New York, NY, USA, Association for Computing Machinery, p. 23–28 (online), DOI: 10.1145/3486949.3486966 (2021).

[7] Obara, M., Akinobu, Y., Kajiura, T., Takano, S. and Kuramitsu, K.: A Preliminary Report on Novice Programming with Natural Language Translation, *IFIP WCCE 2022: World Conference on Computers in Education* (2022).

[8] Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A. and Ma, S.: CodeBLEU: a Method for Automatic Evaluation of Code Synthesis, *CoRR*, Vol. abs/2009.10297 (online), available from ⟨https://arxiv.org/abs/2009.10297⟩ (2020).

[9] Papineni, K., Roukos, S., Ward, T. and Zhu, W.-J.: Bleu: a Method for Automatic Evaluation of Machine Translation, *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, Philadelphia, Pennsylvania, USA, Association for Computational Linguistics, pp. 311–318 (online), DOI: 10.3115/1073083.1073135 (2002).

[10] Lin, C.-Y. and Och, F. J.: Automatic Evaluation of Machine Translation Quality Using Longest Common Subsequence and Skip-Bigram Statistics, *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, Barcelona, Spain, pp. 605–612 (online), DOI: 10.3115/1218955.1219032 (2004).

[11] Yujian, L. and Bo, L.: A Normalized Levenshtein Distance Metric, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 29, pp. 1091–1095 (2007).

[12] Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K.: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, *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)*, Minneapolis, Minnesota, Association for Computational Linguistics, pp. 4171–4186 (online), DOI: 10.18653/v1/N19-1423 (2019).

[13] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W. and Liu, P. J.: Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer, *J. Mach. Learn. Res.*, Vol. 21, pp. 140:1–140:67 (online), available from ⟨http://jmlr.org/papers/v21/20-074.html⟩ (2020).

[14] Sun, Z., Zhu, Q., Xiong, Y., Sun, Y., Mou, L. and Zhang, L.: Treegen: A tree-based transformer architecture for code generation, *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34, No. 05, pp. 8984–8991 (2020).

[15] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K. et al.: Google's neural machine translation system: Bridging the gap between human and machine translation, *arXiv preprint arXiv:1609.08144* (2016).

[16] Li, Yujia et. al.and Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D., Sutherland Robson, E., Kohli, P., de Freitas, N., Kavukcuoglu, K. and Vinyals, O.: Competition-Level Code Generation with AlphaCode, *arXiv preprint arXiv:2203.07814* (2022).

[17] Beau, N. and Crabbé, B.: The impact of lexical and grammatical processing on generating code from natural language, *Findings of the Association for Computational Linguistics: ACL 2022*, Dublin, Ireland, Association for Computational Linguistics, pp. 2204–2214 (online), DOI: 10.18653/v1/2022.findings-acl.173 (2022).

[18] 相馬菜生，梶浦照乃，小原百々雅，倉光君郎：CoNaLa チャレンジ: 言語生成モデル T5 によるコード生成，2022 年並列／分散／協調処理に関するサマー・ワークショップ (SWoPP2022). 情報処理学会 第 140 回プログラミング研究発表会 (PRO140) (2021).

[19] Yin, P., Deng, B., Chen, E., Vasilescu, B. and Neubig, G.: Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow, *International Conference on Mining Software Repositories*, MSR, ACM, pp. 476–486 (online), DOI: https://doi.org/10.1145/3196398.3196408 (2018).

[20] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D. and Zhou, M.: CodeBERT: A Pre-Trained Model for Programming and Natural Languages (2020). cite arxiv:2002.08155Comment: Accepted to Findings of EMNLP 2020. 12 pages.

[21] Tran, N., Tran, H., Nguyen, S., Nguyen, H. and Nguyen, T. N.: Does BLEU Score Work for Code Migration?, *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, IEEE Press, p. 165–176 (online), DOI: 10.1109/ICPC.2019.00034 (2019).

[22] Karaivanov, S., Raychev, V. and Vechev, M.: Phrase-Based Statistical Translation of Programming Languages, *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2014, New York, NY, USA, Association for Computing Machinery, p. 173–184 (online), DOI: 10.1145/2661136.2661148 (2014).

[23] Roziere, B., Lachaux, M.-A., Chanussot, L. and Lample, G.: Unsupervised Translation of Programming Languages, *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, Curran Associates Inc. (2020).