一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report

LLMを用いたソースコード改変タスクにおける Fine-Tuningの実験的調査に向けて

數﨑　大樹†　松本　真佑†　楠本　真二†
安田　和矢††　伊藤　信治††　代　吉楠††

† 大阪大学大学院情報科学研究科
†† 株式会社 日立製作所
E-mail: †{d-kazusk,shinsuke}@ist.osaka-u.ac.jp

あらまし　ソフトウェア開発に関する様々なタスクに対して，大規模言語モデル（Large Language Model; LLM）を用いた支援や自動化が可能となってきた．Fine-Tuning（FT）とは事前学習済みモデルに対して，特定タスクに特化させる再学習手法である．FT の適用により，自然言語やソースコードに対する汎用的な知識を持つ事前学習済みモデルから，特定タスクに特化したモデルを獲得できる．FT の適用においては，再学習のため再学習データセットの構築が欠かせない．しかしながら，ソフトウェア開発タスクにおいてどのような再学習データセットをどの程度用意すべきかは明らかではない．本研究の長期的な目的は，LLM を用いたソフトウェア開発における高精度かつ高効率な FT 適用方法の獲得である．本研究会原稿では，ソースコードの改変というタスクに着目し，簡易的な再学習データセットを用いて FT 済みモデルと事前学習済みモデルの性能を実験的に比較する．

キーワード　LLM，Fine-Tuning，再学習データセット，データセットの量，データセットの質

1. はじめに

ソフトウェア開発に関する様々なタスクに対して，大規模言語モデル（Large Language Model; LLM）を用いた支援や自動化が可能となってきた．バグの自動修正 [1] や自動リファクタリング [2] といったソースコードの一部を改変するタスクだけでなく，プログラム生成 [3] やテスト生成 [4] といったソースコード全体を生成するタスクも研究が進められている．また，自然言語とソースコードの両方を学習したモデルも多数登場しており，ソースコード要約 [5] やコードレビュー [6] 等の自然言語とソースコードを扱うタスクも LLM の支援可能な領域となりつつある．

LLM に対する性能改善方法の 1 つとして，Fine-Tuning（FT）が広く知られている [7]．FT とは，事前学習済みモデルを特定のタスクに特化させる再学習手法である．一般的に事前学習済みモデルを構築する際には，大量のデータセットを用いてモデルをゼロから学習させる必要がある．他方，FT では特定のタスクの再学習データセットを用いて，事前学習済みモデルが持つパラメタの再更新を行う．FT の適用によって，大量データから得られた汎用的な知識を持つ事前学習済みモデルに対して，特定タスクの解決に特化した専門的な知識の埋め込みが可能となる．

LLM を活用するソフトウェア開発タスクにおいて，高性能なモデルを獲得できる FT の適用方法は明らかではない．特に，再学習データの量や質は FT 後モデルの性能を左右する重要な要素であり [8]，また FT 利用者が決定すべき一種のハイパパラメタである．数多くのデータを確保するほどオーバフィッティングを低減できる一方で，データの構築に要する人的コストの増大にも繋がる．また再学習データに含まれるノイズの量はモデルの性能低下に繋がる [9]．LLM の学術領域では FT に用いる再学習データの量と質に関して様々な研究が実施されているものの，ソースコードを扱う LLM に対する調査は行われていない．

ここでソフトウェア開発タスクを構成する様々なタスクのうち，ソースコードの改変というタスクに着目する．このタスクでは，開発者の改変指示に基づいて既存のソースコードを書き換える．すなわち入力は自然言語で記述されたテキスト，及び改変前ソースコードであり，出力は改変後ソースコードとなる．このソースコード改変タスクは，開発の下流工程のほとんどを占める極めて一般的かつ継続的なタスクである．よって，このタスクへの LLM の適用，及びその性能の改善はソフトウェア開発における 1 つの重要なトピックであると考える．

本研究の目的は，LLM を用いたソースコード改変タスクにおける高性能かつ高効率な FT 適用方法の獲得である．本研究会原稿では，この目的を達成するために必要となる実験の方針について考える．特に FT に用いる再学習データセットの量と質，及びそれらをどう制御すれば高性能かつ高効率な FT に繋がるかを議論する．またこの再学習データセットの構築方法に関しても議論を行う．また小規模な予備実験の結果を報告する．この予備実験では，OSS のコミットログから 109 件のソースコード改変データを作成し，実際に FT を適用した場合と適用しない場合の性能を比較する．

2. 準　　備

2.1 Fine-Tuning

Fine-Tuning（FT）とは，事前学習済みモデルに対する性能改善方法の 1 つである [10]．FT では，事前学習済みモデルのパラメタを更新することによって，汎用的な能力を有する事前学習済みモデルを特定のタスクへの特化が可能になる．

FT には，モデルの全パラメタを更新する Full Fine-Tuning (FFT) とモデルの一部のパラメタのみを更新する Parameter-Efficient Fine-Tuning (PEFT) [11] の 2 種類がある．FFT は事前学習済みモデルの全てのパラメタを更新する手法である．しかし，FFT は計算資源の面でコストが大きく，特に数十億規模のパラメタを持つ大規模モデルに対しての適用は困難である [12]．これに対して，PEFT はパラメタの大部分を変更せず，その一部のみを更新する [13]．これによって，必要とする計算資源を大幅に削減しつつ [14]，FFT と同等あるいはそれ以上の性能を達成できる．よって，事前学習済みモデルを特定のタスクに適応させる場合，より実現可能なアプローチとなっている．

2.2 ソースコード改変タスクと Fine-Tuning

ソフトウェア開発の支援を目的として，FT の効率的な適用方法に関する研究が広く実施されている [15]〜[18]．具体的には，バグの自動修正 [15] やソースコード補完 [16] などのソースコードの一部のみを変更するタスクだけでなく，ソースコード生成 [17] やテスト生成 [18] などソースコード全体を生成するタスクに対しても研究が行われている．

ソフトウェア開発を構成する様々なタスクのうち，本稿ではソースコード改変タスクに焦点を当てる．本稿におけるソースコード改変タスクという言葉は次の入出力を成す作業の意味で用いる．

- 入力：改変前ソースコード, 自然言語の改変指示
- 出力：改変後ソースコード

ソースコード改変はプログラミング作業の大部分を占める実践的な活動であり，ソフトウェア開発の現場で頻繁に発生する．そのため，このタスクに LLM を適用することは，開発効率の向上に寄与する有効なアプローチと考えられる．

これまでにソフトウェア開発のおけるタスクでは FT を用いた研究は数多く行われている [19]．しかしながら，筆者らの知る限り，ソースコード改変タスクに特化して LLM の性能を体系的に検証した研究は存在しない．

3. Research Question

本研究の目的は，ソースコード改変タスクにおける高精度かつ高効率な FT の適用方法の獲得である．高精度なモデルの獲得は当然ながら，効率的な学習の実施も重要な指針である．同程度の精度のモデルが得られるのであれば，人的コストが少なく，より少ない量かつ短時間で FT を完了できるデータセットが望ましい．この目的を達成するために，本研究では以下のような Research Question を設定する．

RQ1：再学習データセットの量はどうあるべきか

FT は事前学習に比べて少量のデータで実施できるが，再学習データセットの構築には依然として人的コストが発生する [20] [21]．したがって，高い精度を維持できるデータ量の把握が重要である．本研究ではこの観点から，精度を損なうことなく FT を行うために必要なデータセットの量を調査する．

RQ2：再学習データセットの質はどうあるべきか

FT においては，データの量だけではなくデータの質もモデル性能に大きく影響を与える要因である [22]．ソースコード改変タスクにおけるデータの質としては，改変指示の質やソースコードの構造的な質が考えられる．改変指示の質はタスクの精度に強く寄与すると考えられるが，高い抽象度で端的な指示が適しているのか，詳細かつ具象的な指示が適しているのかは不明である．また，ソースコードの構造的な質もデータセットにおける重要な要素である．ソースコードの構造的な質としては，データセット全体の多様性，及びソースコードの範囲の 2 点が考えられる．データセット全体の多様性はモデルが学習するデータの幅に影響を与える．タスク適用先のプロジェクトから収集された多様性の低い特化型のデータを用いるべきか，様々なプロジェクトから収集された多様性の高い汎用的なデータを用いるべきかは明らかになっていない．一方，ソースコードの範囲はデータセットのノイズや文脈情報の量に影響を与える．変更が発生したメソッド群のみが含まれるノイズの少ないデータを用いるべきか，変更が発生したクラス全体が含まれる豊かな文脈情報を持つデータを用いるべきかも明らかとすべき重要な視点である．

なお，本研究で扱うソースコード改変はプログラムの機能的な振る舞いを変更するものに限定する．すなわち，ソースコードの可読性や構造を改善することを目的とするリファクタリングは対象外とする．リファクタリングに関しては，LLM を用いない古典的で決定的な自動化手法 [23] が確立されているためである．

4. 実 験 方 針

本節では前節で述べた 2 つの RQ に答えるための実験方針について議論する．図 1 に実験全体の流れを示す．実験は図中に示す 7 ステップで構成される．以降，各ステップの処理内容，及びステップ内で発生する検討項目について議論する．図中においては，実線矢印が各ステップであり，紫の吹き出しが本研究で実験的に比較すべき選択肢，灰色の吹き出しは LLM 自体関係，つまり本研究の主眼ではない選択肢である．

Step 1. コミットの抽出とフィルタリング

再学習データの構築のために，実際のソフトウェア開発リポジトリから改変タスクを表すデータを回収する．バージョン管理において，1 つのコミットにはソースコードに対する意味のある改変のまとまりとすべきというプラクティスが存在する [24]．すなわち，個々のコミットが 1 つのソースコード改変タスクであるとみなせる．また，コミットにはメッセージが付与されている．コミットメッセージの内容は多くの場合動詞で開始する文章であり，改変指示とみなすこともできる．

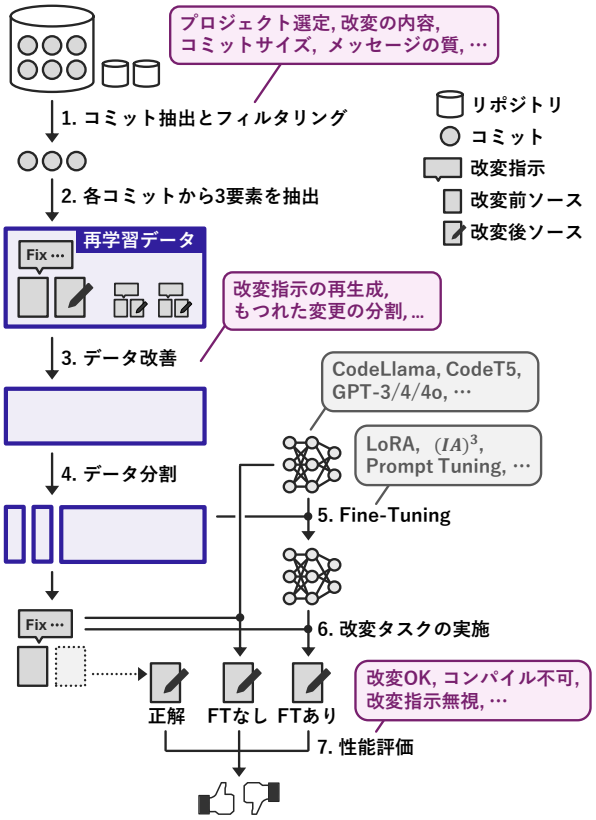

図 1: 実験の流れ

回収するコミットには一定のフィルタリングを適用すべきである．第一にプロジェクトのフィルタリングが必要である．採用しているプログラミング言語の選定は当然ながら，そのプロジェクトの性質や開発コミュニティの違い等も考慮に入れるべきである．また，リポジトリ内のコミットに対するフィルタリングも必要である．実際の開発の際に記録されたコミットは，改変内容や改変サイズが一定ではなく，コミットメッセージが改変の内容を適切に表しているとも限らない．上記のような基準を設けてコミットを収集すべきか，あるいは基準を設けずに全コミットを再学習に用いるべきかは実験によって明らかにすべき一つのパラメタである．基準を設ける場合，再学習データ数は減るもののデータの品質は改善される．設けない場合は完全にその逆であり，どちらを採用すべきかを明らかにすべきである．

なお，振る舞いを変更するコミットのみを抽出する方法としては，様々な方法が考えられる．軽量な手段としては，コミットメッセージに基づいて振る舞いの変更が発生しないコミットを排除する方法が考えられる．またテストが同時に変更されているコミットは，振る舞いが変更されている可能性が高い．よってコミットで変更されたファイルを確認するという方法も存在する．より精度の高い手段はテストの実行である．改変前ソースコードからテスト自動生成 [3] を適用し，改変後ソースコードに適用すれば，その機能の等価性を自動で判断できる．

Step 2. 各コミットから 3 要素を抽出

次に収集したコミットから，再学習データに用いる 3 つの要素を抽出し，再学習データセットを作成する．3 つの要素とは，自然言語で記述された改変指示，改変前ソースコード，及び改変後ソースコードである．ソースコード以外の改変ファイル（README.md や LICENSE 等）は本研究で扱うソースコード改変タスクの範囲外であり，抽出する必要はない．1 コミットで複数のソースコードが改変されていた場合は，その全てのソースコードの前後を回収する．我々の扱う改変タスクとは，1 ファイルに限定した処理ではないためである．

Step 3. データ改善

Step 2 で生成した生の再学習データを改善する．本ステップは実験における必須ステップではない．データ改善には様々な方法が考えられる．1 つは改変指示の改善である．コミットに付記されるメッセージの 44% はその内容を適切に表していないという報告も存在する [25]．改変指示が適切でない場合，LLM による高い精度での改変は期待できない．

この問題に対する解決策として，LLM を用いた改変メッセージの再生成が考えられる．改変前後のソースコードを LLM に与えその差分を説明させれば，改変指示の再生成が可能である．再生成された改変指示と元の改変指示を比較すれば，その内容の適切さもある程度判定可能である．

また 1 コミットの中に複数の改変が混在している場合がある．これはもつれた変更と呼ばれる．もつれた変更を独立した変更群に分解する手法は存在しており [26]，これを用いることで単一の意図のみで構成された変更を得ることが可能である．

Step 4. データ分割

改善された再学習データを FT のために分割する．典型的な分割割合は FT の訓練データ，FT の検証データ，評価に用いるテストデータの割合はそれぞれ 80%，15%，5% である [27]．

Step 5. Fine-Tuning の適用

再学習データセットを用いて事前学習済みモデルに FT を適用する．事前学習済みモデルには複数の選択肢が考えられるが，本研究における選定の基準は以下の 2 点である．1 点目はソースコードを事前学習していること，2 点目はモデルが OSS として公開されていることである．この要件を満たしているモデルとしては，CodeLlama [28]，CodeT5 [29]，GPT シリーズ [30] [31] [32]，Llama [33] 等がある．また，FT にも複数の選択肢が考えられるが，本研究においては他の実験で広く用いられている PEFT を採用する．具体的な PEFT の手法としては LoRA [34]，$(IA)^3$ [35]，Prompt Tuning [36] が挙げられる．

なお，事前学習済みモデルや FT の選定は本研究の主眼ではない．この理由としては，本研究が FT に用いるデータ作成に焦点を当てているためである．

Step 6. 改変タスクの実施

FT が適用されたモデル（FT あり）と事前学習済みモデル（FT なし）の両方に対して，改変タスクを実施する．再学習に用いなかったテストデータ中の 1 改変事例を取り出し，その中の改変指示と改変前ソースコードを LLM に与えるプロンプトとする．これをテストデータの全改変事例に適用し，複数の様々な改変事例に対する性能を得る．タスク実施の際のプロンプトとしては，zero-shot 学習 [37] [38] や few-shot 学習 [39] [40]

など様々なプロンプトエンジニアの方法を採用できる．

なお，プロンプトの適切な設計に関しては本研究の主眼ではない．これも Step 5 と同様の理由である．

Step 7. 性能評価

様々な条件で得られた FT ありを比較することで，高精度かつ高効率な FT の適用方法を確認する．また，FT なしをベースラインとし，どの再学習データセットの作成方法が精度と効率の面で優れているかを調べる．精度は，改変後ソースコードが元の改変後ソースコード（正解）と同じ機能を提供している割合，つまり指示通りに改変できた割合によって計測する．効率は FT に要する計算コストだけでなく，FT に用いた再学習データの構築コストも考慮すべきである．

続いて，LLM が出力する改変後ソースコードの正しさの分類とその分類方法を考える．改変後ソースコードは次の 4 種類に分類できる．

指示を満たす（正解）：LLM による改変が成功したケースである．LLM が生成した改変後ソースコードが，元の改変後ソースコードと同じ機能を提供する場合，このケースに該当すると判断する．その確認にはテストを用いた自動判定が可能である．テストを用いることで，変数名やインデントの深さといったテキスト上の違いや，for と while のようなプログラム構造の違いなどを許容した柔軟な正誤判定が可能である．テストの生成は，古典的な自動テスト生成 [41] や LLM によるテスト生成 [4] 等を利用できる．

指示無視（不正解）：LLM で散見されるケースである．LLM がプロンプトで与えた改変指示を無視し，期待される作業とは全く別の作業を行っている場合である．具体的には，改変という指示に対して要約やリファクタリングを行うケースなどが挙げられる．LLM の出力にソースコードが含まれていない場合は，改変指示を満たさない要約のようなケースと判断できる．また正解のケースと同様，テストによって改変が適用されていないかも自動判定できる．改変前ソースコードから生成した全てのテストに通過する場合，改変を行っていない（リファクタリング等）と判断できる．

コンパイル不可（不正解）：LLM が生成した改変後ソースコードに構文的な誤りを含むケースである．改変前ソースコードはコンパイル可能であるため，LLM が構文的な誤りを含めてしまったと判断できる．このケースの確認はコンパイルを実行するのみで確認できる．

指示を満たさない（不正解）：上記 3 つに含まれないケースを，指示を満たさない不正解だと捉える．指示無視とコンパイル不可の 2 ケースと比較すると，少なくとも改変指示を満たすよう取り組んだケースであると考えられる．確認方法は元データの改変後ソースコードからテストを生成し，LLM が生成した改変後ソースコードに適用すれば良い．一部のテストが失敗すれば，改変指示を満たしていないと見なすことができる．

5. 予 備 実 験

前節で述べた実験方針の妥当性の確認のために予備実験を実施した．予備実験では簡易的な 1 種類のデータセットしか用意できなかったので，FT あり同士の比較は行っておらず，FT ありと FT なしの比較のみを行っている．本研究会原稿では，この予備実験の設計，及びその結果について報告する．

5.1 予備実験の設計

5.1.1 再学習データセットの構築

再学習データセットは，GitHub の Java プロジェクトからコミットを収集することで構築した．プロジェクトの選定基準は，Java のテストコードを含んでいることである．この選定基準を満たしたプロジェクトを抽出し，コミットを収集した．コミットの採用条件は，以下の 3 点とした．

- 1 コミットにおけるソースコードの変更が 1 クラス内に閉じていること．データセット構築を容易にするため．
- 1 コミットにおけるソースコードの変更が 1,000 行以下であること．極端に大規模な変更を除外し，改変前後の対応関係を明確に保つため．
- ソースコードとそれに対応するテストコードの両方が変更されていること．振る舞いの変更にはテストコードの修正が伴うという前提による．

以上の要件を満たしたコミットは 109 件であった．さらに，改変指示にはコミットメッセージをそのまま使用した．この理由は，コミットメッセージが改変指示として有効に機能するかを検証するためであり，フィルタリングによる人為的なバイアスを避ける意図がある．

5.1.2 事前学習済みモデル，Fine-Tuning の選定

事前学習済みモデルは meta-llama/Llama-3.2-1B-Instruct を採用した．これは，前節 Step5 で述べたモデルの条件を満たしており，予備実験で使用した GPU[1]で実行可能なためである．FT は LoRA [34] を採用した．これは，PEFT の中でも既存研究で広く採用されている手法であることが示されている [19] ことを理由とする．

5.1.3 実 験 題 材

予備実験ではデータ数が少量のため，データ分割を行わず，収集したデータを全て FT の学習に使用した．そのため，性能評価専用のデータとして，難易度が異なる 2 種類の実験題材を別途用意した．

図 2 は 1 つ目の題材である．改変前ソースコードには 2 整数の加算結果を標準出力する Java プログラム，改変指示には 2 整数の加算結果を返す calculate メソッドを 2 整数の乗算結果を返すように変更する指示を含むプロンプトである．改変後ソースコードとして期待されるのは，calculate メソッドの返り値を `return num1 + num2;` から `return num1 * num2;` に変更したソースコードである．

図 3 は 2 つ目の題材である．改変前ソースコードには配列の要素を順方向に標準出力する Java プログラム，改変指示には配列の要素を逆順に表示するように変更する指示を含むプロンプトである．改変後ソースコードとして期待されるのは，printAllItem メソッドの `System.out.println(item[i]);`

(注1)：NVIDIA GeForce RTX 3070 Ti

````
Change calculate method to return the product of two
     integers.
```java
public class Main {
  public static void main(String args[]) {
    int sum = calculate(3, 4);
    System.out.println(sum);
  }

  public static int calculate(int num1, int num2) {
    return num1 + num2;
  }
}
```
````

図 2: LLM に与えた題材 1

````
Change printAllItems method to print the array in
     reverse order.
```java
public class Main {
  public static void main(String args[]) {
    String items[] = {"apple", "banana", "cherry"};
    printAllItems(items);
  }

  public static void printAllItems(String[] items) {
    for (int i = 0; i < items.length; i++) {
      System.out.println(items[i]);
    }
  }
}
```
````

図 3: LLM に与えた題材 2

を `System.out.println(item[items.length-i-1]);` に変更したソースコードである．

5.1.4 性 能 評 価

FT あり，FT なしの性能を定量的に評価するために，各題材に対して 100 回ずつソースコード改変タスクを実施した．この 100 回という試行回数は，LLM から得られる出力の揺らぎを十分に観測し，統計的に安定した傾向を得ることを目的として設定した．

得られた改変後ソースコードは，前節 Step7 で示した指示を満たす，指示無視，コンパイル不可，及び指示を満たさないの 4 種類に分類し，試行回数に対する割合をそれぞれ算出した．これら割合を FT あり，FT なしで比較することにより，モデルの性能を評価した．

5.2 予備実験の結果

FT あり，FT なしにそれぞれ図 2，図 3 に示す題材を与えた．図 4，図 5 は各題材に対する指示無視，コンパイル不可，指示を満たさない，及び指示を満たすの割合を示したグラフである． 比較的簡単な題材 1 においては，FT ありと FT なしで指示を満たす割合に大きな差は見られなかった．一方で，題材 1 と比べて難易度が高い題材 2 では FT ありが FT なしよりも指示を満たすの割合が高く，指示無視の割合が低かった．また，題材 1，題材 2 のいずれにおいても FT ありと FT なしでコンパイル不可の割合に大きな差は見られなかった．

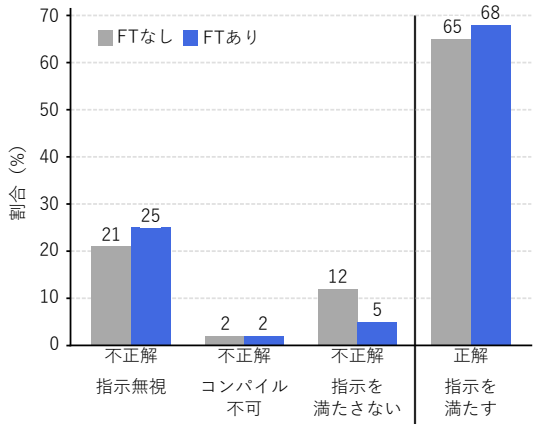

図 4: 題材 1 に対する実験結果

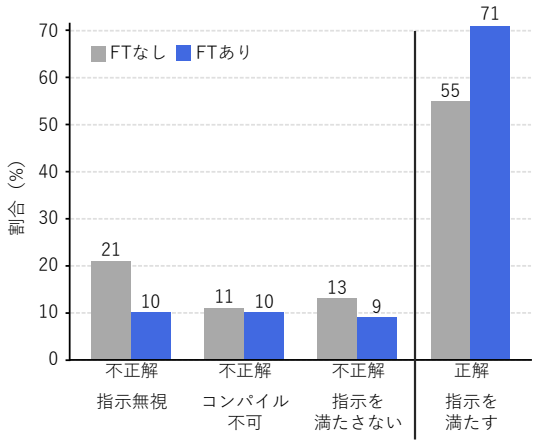

図 5: 題材 2 に対する実験結果

5.3 考　　察

このような結果になった理由としては，再学習データセットには改変指示に従ってソースコードを改変する事例のみが含まれており，コンパイルエラーを修正する事例が含まれていなかったことに起因すると考えられる．まず，LLM は事前学習の段階で LLM は Java の構文を十分に理解している．次に，改変指示に従ってソースコードを改変する能力は FT によって向上したので，改変指示を満たすソースコードの割合は向上した．その分指示無視及び改変指示を満たさないソースコードの割合が減少したと考えられる．しかし，コンパイルエラーを修正する学習を行っていないため，FT による性能向上は見られなかったと推察される．

6. おわりに

本研究の目的はソースコード改変タスクにおける高精度かつ高効率な FT 適用方法の獲得である．本稿では，目的達成に向けた実験設計を議論し，この実現に向けた予備実験を実施した．結果としては，題材 2 のような難しい題材を与えたときに大幅な精度の向上が見られた．

今後取り組むべき課題として，以下の 2 点を挙げる．1 点目はデータセットの拡充である．データセットの量に関しては，引き続きデータ収集を進め，既存研究 [15] で用いられた 30k 件程度の規模を目指す．データセットの質に関しては，改変指示及びソースコードの質の向上を図り，複数のデータセットを構築することで，データの質がモデルの性能に与える影響について検証を行う．2 点目は再学習データセットの構築コストの定量化である．特に，フィルタリング工程のコストを定量化することで，全体の構築効率を算出することが可能になる．これにより，作業量と品質管理に必要な人的コストを明確にでき，現実的に作成可能なデータセットの量や質を把握することにつながる．

謝辞 本研究の一部は，JSPS 科研費（JP25K15056, JP25K03102, JP24H00692）による助成を受けた．

文　献

[1] M. Jin, S. Shahriar, M. Tufano, X. Shi, et al., “InferFix: End-to-end program repair with LLMs,” In Proc. ACM Joint European Softw. Eng. Conf. and Symp. the Foundations of Softw. Eng., pp.1646–1656, 2023.

[2] J. Cordeiro, S. Noei, and Y. Zou, “An empirical study on the code refactoring capability of large language models,” arXiv:2411.02320 [cs.SE], 2024.

[3] Y. Dong, X. Jiang, J. Qian, T. Wang, et al., “A survey on code generation with llm-based agents,” arXiv:2508.00083 [cs.SE], 2025.

[4] Z. Yuan, M. Liu, S. Ding, K. Wang, et al., “Evaluating and improving ChatGPT for unit test generation,” J. Proc. ACM Softw. Eng., vol.1, no.FSE, pp.1–24, 2024.

[5] T. Ahmed and P. Devanbu, “Few-shot training LLMs for project-specific code-summarization,” In Proc. Automated Softw. Eng., pp.1–5, 2023.

[6] J. Lu, L. Yu, X. Li, L. Yang, and C. Zuo, “LLaMA-Reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning,” In Proc. Int’l Symp. Software Reliability Engineering, pp.647–658, 2023.

[7] A. Zhang, Z.C. Lipton, M. Li, and A.J. Smola, Dive into deep learning, Cambridge University Press, 2023.

[8] D. Zha, Z.P. Bhat, K.-H. Lai, F. Yang, et al., “Data-centric artificial intelligence: A survey,” arXiv:2303.10158 [cs.LG], 2023.

[9] S. Ahn, S. Kim, J. Ko, and S.-Y. Yun, “Fine tuning pre trained models for robustness under noisy labels,” arXiv:2310.17668 [cs.LG], 2023.

[10] G.W. Anderson and D.J. Castaño, “Measures of fine tuning,” J. Physics Letters B, vol.347, no.3, pp.300–308, 1995.

[11] Z. Fu, H. Yang, A.M.-C. So, W. Lam, L. Bing, and N. Collier, “On the effectiveness of parameter-efficient fine-tuning,” arXiv:2211.15583 [cs.CL], 2022.

[12] R. Patil and V. Gudivada, “A review of current trends, techniques, and challenges in large language models (llms),” J. Applied Sciences, vol.14, no.5, pp.1–2074, 2024.

[13] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, et al., “Parameter-efficient transfer learning for nlp,” arXiv:1902.00751 [cs.LG], 2019.

[14] R. Shuttleworth, J. Andreas, A. Torralba, and P. Sharma, “Lora vs full fine-tuning: An illusion of equivalence,” arXiv:2410.21228 [cs.LG], 2025.

[15] G. Li, C. Zhi, J. Chen, J. Han, and S. Deng, “Exploring parameter-efficient fine-tuning of large language model on automated program repair,” arXiv:2406.05639 [cs.SE], 2024.

[16] B. Liu, C. Chen, Z. Gong, C. Liao, et al., “MFTCoder: Boosting code llms with multitask fine-tuning,” In Proc. KDD, pp.5430–5441, 2024.

[17] S. Ayupov and N. Chirkova, “Parameter-efficient finetuning of transformers for source code,” arXiv:2212.05901 [cs.CL], 2022.

[18] D. Goel, R. Grover, and F.H. Fard, “On the cross-modal transfer from natural language to code through adapter modules,” arXiv:2204.08653 [cs.CL].

[19] S. Afrin, M.Z. Haque, and A. Mastropaolo, “A systematic literature review of parameter-efficient fine-tuning for large code models,” arXiv:2504.21569 [cs.SE], 2025.

[20] J.C. Chang, S. Amershi, and E. Kamar, “Revolt: Collaborative crowdsourcing for labeling machine learning datasets,” In Proc. Conf. Human Factors in Computing Systems, pp.2334–2346, 2017.

[21] T. Kulesza, S. Amershi, R. Caruana, D. Fisher, and D. Charles, “Structured labeling for facilitating concept evolution in machine learning,” In Proc. Conf. Human Factors in Computing Systems, pp.3075–3084, 2014.

[22] R. Rejeleene, X. Xu, and J. Talburt, “Towards trustable language models: Investigating information quality of large language models,” 2024.

[23] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: improving the design of existing code, Addison-Wesley Longman Publishing Co., Inc., 1999.

[24] M. diBiase, M. Bruntink, A. vanDeursen, and A. Bacchelli, “The effects of change decomposition on code review—a controlled experiment,” J. PeerJ Computer Science, vol.5, pp.1–193, 2019.

[25] Y. Tian, Y. Zhang, K.-J. Stol, L. Jiang, and H. Liu, “What makes a good commit message?,” In Proc. 44th ICSE, pp.2389–2401, 2022.

[26] M. Wang, Z. Lin, Y. Zou, and B. Xie, “CoRA: Decomposing and describing tangled code changes for reviewer,” In Proc. Int’l Conf. on Automated Softw. Eng., pp.1050–1061, 2019.

[27] V.R. Joseph, “Optimal ratio for data splitting,” J. Statistical Analysis and Data Mining: The ASA Data Science Journal, vol.15, no.4, pp.531–538, 2022.

[28] B. Rozière, et al., “Code Llama: Open foundation models for code,” arXiv:2308.12950 [cs.CL], 2024.

[29] Y. Wang, W. Wang, S. Joty, and S.C.H. Hoi, “CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” arXiv:2109.00859 [cs.CL], 2021.

[30] L. Floridi and M. Chiriatti, “GPT-3: Its nature, scope, limits, and consequences,” J. Minds Machines, vol.30, no.4, pp.681–694, 2020.

[31] J.A. Baktash and M. Dawodi, “GPT-4: A review on advancements and opportunities in natural language processing,” arXiv:2305.03195 [cs.CL], 2023.

[32] OpenAI, A. Hurst, et al., “GPT-4o system card,” arXiv:2410.21276 [cs.CL], 2024.

[33] A. Grattafiori, et al., “The llama 3 herd of models,” arXiv:2407.21783 [cs.AI], 2024.

[34] E.J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, et al., “LoRA: Low-rank adaptation of large language models,” J. ICLR, vol.1, no.2, pp.1–3, 2022.

[35] H. Liu, D. Tam, M. Muqeeth, J. Mohta, et al., “Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning,” arXiv:2205.05638 [cs.LG], 2022.

[36] B. Lester, R. Al-Rfou, and N. Constant, “The power of scale for parameter-efficient prompt tuning,” arXiv:2104.08691 [cs.CL], 2021.

[37] H. Larochelle, D. Erhan, and Y. Bengio, “Zero-data learning of new tasks,” In Proc. AAAI, pp.646–651, 2008.

[38] C.H. Lampert, H. Nickisch, and S. Harmeling, “Attribute-based classification for zero-shot visual object categorization,” J. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol.36, no.3, pp.453–465, 2014.

[39] L. Fei-Fei, R. Fergus, and P. Perona, “One-shot learning of object categories,” J. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol.28, no.4, pp.594–611, 2006.

[40] M. Fink, “Object classification from a single example utilizing class relevance metrics,” In Proc. NeurIPS, pp.1–17, 2004.

[41] G. Fraser and A. Arcuri, “Evosuite: automatic test suite generation for object-oriented software,” In Proc. Foundations of Softw. Eng., pp.416–419, Association for Computing Machinery, 2011.