東京大学
情報理工学系研究科 創造情報学専攻
修士論文

LLM のコード翻訳を応用した異言語慣用句の混入に対する修正候補の提示

A system for proposing correction candidates for different-language idiomatic code by LLM code translation

宮原 和也
Miyahara Kazuya

指導教員　千葉 滋 教授

2024 年 1 月

概要

複数言語を使い分けて開発する環境では，言語間の慣用句の違いによって開発者が混乱することがあるため，異言語の慣用句の使用を認識して，その修正候補を提示する入力支援の開発が望まれる．本研究では，大規模言語モデル (LLM) を用いたコード翻訳に，異言語の慣用句の混入を判定する処理と，翻訳前と翻訳後のコードから慣用句部分を抽出する処理を組み合わせて，この入力支援を開発する手法を提案する．この提案手法によって，LLM のサーバーに大量リクエストすることによって起こる問題，および LLM がコード内の慣用句部分を認識できない問題に対処しつつ，LLM を用いた高精度のコード翻訳を入力支援に適応することができる．本手法によって実装した入力支援に対して，異言語の慣用句の混入を判定する性能を検証した．また，入力支援が提示する修正候補の質に対しても検証を行った．

Abstract

In environments where multiple languages are used for development, developers may encounter confusion due to differences in idiomatic expressions across languages. Therefore, there is a need for input assistance that recognizes the use of idiomatic expressions in foreign languages and suggests correction candidates. This study proposes a method for developing such input assistance using Code Translation with Language Model (LLM), which combines processes for identifying the presence of foreign language idioms and extracting idiomatic expression segments from both pre-translated and post-translated code. This proposed method addresses issues arising from the high volume of requests to LLM servers and the inability of LLM to recognize idiomatic expression parts within the code. It enables the application of high-precision code translation using LLM as input assistance while mitigating these problems. The implemented input assistance using this method was evaluated for its ability to identify foreign language idioms and its performance in correcting them to the appropriate idiomatic expressions.

目次

第 1 章

はじめに

プログラミングで扱う問題領域が広がったことで，問題領域ごとに別々のプログラミング言語を使う機会も増え，複数のプログラミング言語を使い分ける開発環境が一般的になった．そのような開発環境では，開発者が混乱して，間違った言語の慣用句を書いてしまうことがある．そのため，間違った言語の慣用句の使用を自動的に検知し修正候補をユーザーに提示する，エディタの新しい入力支援が望まれる．

大規模言語モデル (LLM) のコード翻訳を用いれば，このような入力支援が簡単に実現できるように思える．しかし，この入力支援はコード編集中に何度も呼び出す必要があるため，LLM のサーバーに負荷を与え，LLM の使用コストが増大することが考えられる．また，LLM はコード内の異言語の慣用句を翻訳できるが，コード内のどの部分が異言語の慣用句であるか具体的に提示できない．

我々は，前述した入力支援を開発するために，LLM のコード翻訳に，異言語の慣用句が混入しているかを判定する処理と，翻訳前のコードと翻訳後のコードからそれぞれ慣用句部分を抽出する処理を組み合わせる手法を提案する．異言語の慣用句が混入しているかを判定する処理によって，異言語の慣用句が混入していないと判定した場合は，LLM のコード翻訳を行わない．これにより，異言語の慣用句が混入している場合においてのみ LLM を呼び出すことになり，LLM のサーバーの負荷は軽減され，使用コストも減少する．また，LLM のコード翻訳を行った後に慣用句を抽出する処理を加えることで，慣用句単位のコード翻訳を疑似的に行うことができる．

コードに異言語の慣用句が混入しているかを判定する処理は，ユーザーから取得したカーソル周辺の数行のコードを様々な長さのコード片に分割して，それぞれのコード片に対して今書いている言語かどうかの判定をし，それらの結果を突き合わせる方法で行う．コード片が今書いている言語かどうかの判定には，学習済みの LSTM[1] を用いる．コードの翻訳には LLM を用いるが，異言語の慣用句の可能性が高いコード部分だけでなく，ユーザーから取得したコードをすべてモデルに送る．翻訳前のコードと翻訳後のコードからそれぞれ異言語の慣用句と翻訳された慣用句を抽出する処理は，翻訳前のコードと翻訳後のコードを単語単位で比べることでコードの共通部分を見つけ出し，翻訳によってコード内で変わった部分を特定する方法を用いる．

本手法では，プログラムをトークン列に変換する，もしくは単語単位に分割するために頻繁に字句解析を行う．しかし，今書いている言語とは異なる言語で書かれた慣用句も含まれるため，今書いている言語の字句解析器を用いても，うまくいくとは限らない．そのため，今書いている言語が何言語であっても Python の字句解析器を用い，それによる字句解析の結果にヒューリスティックな処理を加えることでトークン化を行う．

入力支援システムを実装するために，コードに異言語の慣用句が混入しているかを判定する処理で用いる LSTM の事前学習を行い，その判定精度を調べた．Python，Java，C++ の 3 つの言語を対象に，コード片がその言語で書かれたものなのか否か判定するタスクを教師つきで LSTM に学習させ，判定精度を調べた．

実装した入力支援システムの評価を行うために，今書いている言語が Python であると仮定し，異言語に対する判定精度および修正精度を検証した．判定精度の検証では，入力支援システム全体でコードに異言語の慣用句が混入しているか否かの判定をどの程度の精度で行えるかを調べた．修正精度の検証では，入力支援システムが出力する修正候補が，我々の用意した理想的な修正候補にどれだけ近いかを調べた．また，実験の前に，異言語の慣用句が混入しているかを判定する処理に用いるパラメータを最適化する実験を行った．

本研究の貢献は大きく以下の 3 つである．

- LLM の API を用いて異言語の慣用句を修正する入力支援を開発するときの課題を発見し，それを克服する手法を提案した．
- 我々が知る限り，異言語の慣用句を修正する入力支援を検証するための最適なデータセットが存在しない．そのため，そのようなデータセットを作成した．
- 提案手法に基づいて異言語の慣用句を修正する入力支援を開発し，その性能を検証した．

本論文の残りの部分の構成について述べる．第 2 章では，我々が目標とする入力支援システムと，それを開発する際の課題について述べる．第 3 章では，その入力支援を開発する手法と，その手法で用いる各処理の手順を述べる．第 4 章では，入力支援システムを実装するうえで必要となる LSTM の事前学習の詳細と，学習後の検証結果について述べる．また，入力支援システム全体に対して行った，異言語の慣用句の混入に対する判定精度の検証と，システムがユーザーに提示する修正内容の検証について実験の詳細を述べる．第 5 章では本研究のまとめと，実験結果を踏まえた今後の課題について述べる．

第 2 章

複数言語を使い分ける開発環境で役立つ入力支援

複数言語を使い分ける開発環境では，プログラマが言語間のコードの書き方の違いによって混乱を起こし，間違った言語で慣用句を書いてしまうことがある．このような場合に，間違った言語の使用を検知し，今書いている正しい言語に翻訳する入力支援が望まれる．

本章では，本研究を理解するうえで必要となる前提知識の説明と，上記に述べた入力支援を開発するための課題を述べる．また，関連研究として，機械学習モデルを用いたコード補完やコード翻訳の研究例や既存の入力支援システムを紹介し，我々の目標とする入力支援との違いを述べる．

2.1 前提知識

本節では，本研究を説明するにあたって必要となる以下の前提知識について述べる．

- ニューラルネットワークモデル
- RNN と LSTM
- 機械学習モデルの学習
- Skip-gram モデル
- 字句解析によるトークン化
- Encoder-Decoder モデルと Transformer
- LLM
- レーベンシュタイン距離

本研究の提案手法では，機械学習モデルである LSTM を用いる．そのため，2.1.1 節では，基本的な機械学習モデルであるニューラルネットワークモデルの仕組みについて述べ，それを発展させた，文章データなど連続的な情報を持つデータに対して機械学習を行うための RNN と，それを応用した LSTM について 2.1.2 節で説明する．また，それらの機械学習モデルの学習方法を 2.1.3 節で説明する．本研究が用いる LSTM は，ソースコードを対象に言語処理

を行うので，機械学習モデルの入力に文章を用いるために，単語それぞれに対応したベクトルを計算する必要がある．その手法の 1 つである SKip-gram モデルについて 2.1.4 節で説明する．その文章がソースコードだった場合に，字句解析を用いてソースコードをトークン化する必要がある理由を 2.1.5 節で述べる．

また，提案手法では，LLM を用いてコード翻訳を行う．そのため，2.1.6 節では，大部分の LLM のベースになっている Transformer と，それに関連した Encoder-Decoder モデルについて説明する．LLM についても 2.1.7 節で解説する．更に，翻訳したコードと元のコードの差異を抽出するために，レーベンシュタイン距離を計算するアルゴリズムを用いる．2.1.8 節では，そのアルゴリズムについて解説する．

2.1.1　ニューラルネットワークモデル

ニューラルネットワークモデルは人間の脳の神経回路のメカニズムを模倣した方法で，コンピュータにデータの処理方法を学習させる機械学習手法の 1 つである．人間の脳に似た層状構造であり，ノード (人工ニューロン) と呼ばれる計算ユニットを相互接続して使用する．

図 2.1 に最も単純なニューラルネットワークモデルを示す．ニューラルネットワークモデルは次に示す 3 つの層でノードを相互接続している．

入力層

外界からのデータを受け取り，各ノードが計算を行って次の層に送る．

中間層

別名，隠れ層とも呼ばれる．入力層や他の中間層からデータを受け取り，各ノードが計算を行って次の層に送る．ニューラルネットワークモデルは，複数の中間層を持つことができる．

出力層

ニューラルネットワークモデルが行ったデータ処理の最終結果を示す．出力層のノード数は対象とする問題の種類によって変わる．例えば，2 値の分類問題であった場合は，出力層は 1 つのノードを持ち，結果は 0 または 1 になる．マルチクラス分類問題の場合は，出力層は複数のノードで構成される．

各ノードの入力値は，前層の接続ノードの出力値，前層の接続ノード間に定義された重み，ノード自身が持つバイアスを用いて計算に用いられる．$(j-1)$ 層にノードが n 個あった場合，j 層の i 層番目のノードの入力値 $x_{j,i}$ は以下に示す式で計算される．

$$x_{j,i} = \sum_{k=1}^{n} w_{i,k}^{j} y_{j-1,k} + b_{j,i} \tag{2.1}$$

ここで，$y_{j,k}$ は j 層目の k 番目のノードの出力値を表し，$w_{i,k}^{j}$ は $(j-1)$ 層目の k 番目のノードと，j 層目の i 番目のノードの間に定義された重みを表す．また，$b_{j,i}$ は j 層の i 番目のノードが持つバイアスである．式 2.1 は，j 層の各ノードの入力値をベクトルの要素に変換

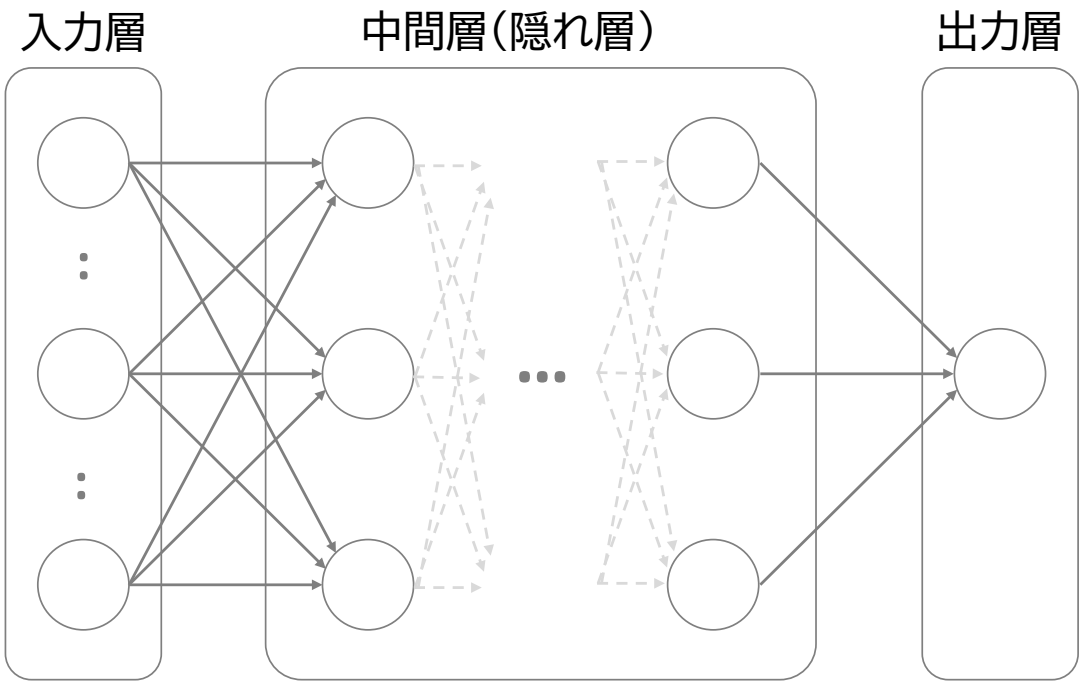


図 2.1. ニューラルネットワークモデルの構造

した入力ベクトル $x_j^l = [x_{j,0}, x_{j,1}, x_{j,2}, x_{j,3}, ...]$ ，j 層の各ノードの出力値をベクトルの要素に変換した出力ベクトル $y_j^l = [y_{j,0}, y_{j,1}, y_{j,2}, y_{j,3}, ...]$ ，j 層の各ノードの持つバイアスのベクトル $b_j^l = [b_{j,0}, b_{j,1}, b_{j,2}, b_{j,3}, ...]$ ，$(j-1)$ 層と j 層の間の重み行列 W_j を用いて，以下の式に変換できる．

$$x_j^l = W_j y_{j-1}^l + b^l \tag{2.2}$$

重み行列 W_j は，行数が j 層のノード数，列数が $(j-1)$ 層のノード数である行列であり，n 行 m 列目の要素は $w_{n,m}^j$ になる．例えば，図 2.2 のような場合においては，重み行列は以下のようになる

$$W_j = \begin{bmatrix} 0.3 & 0.1 & 0.7 \\ 0.2 & 0.5 & 0.2 \end{bmatrix}$$

各ノードの出力値は，ノードの入力値に活性化関数を適用することで計算される．よく使われる活性化関数のいくつかを以下に示す．

シグモイド関数

$$y = \frac{1}{1 + exp(-x)} \tag{2.3}$$

tanh 関数

$$y = \frac{exp(x) - exp(-x)}{exp(x) + exp(-x)} \tag{2.4}$$

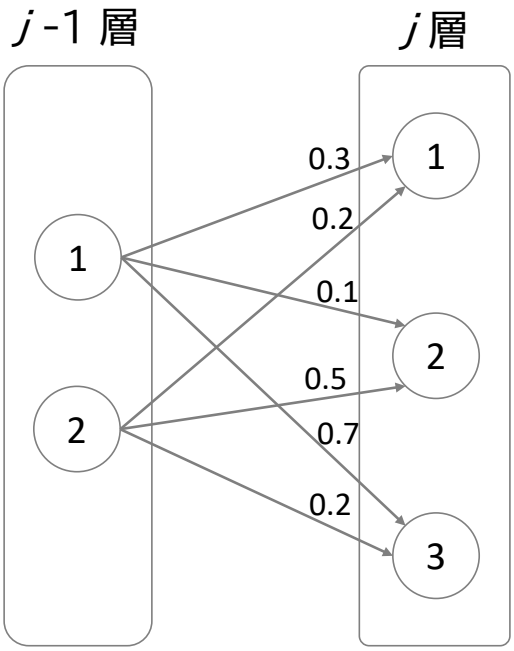


図 2.2. j-1 層と j 層の間の重みの例

ステップ関数

$$y = \begin{cases} 1 & (x > 0) \\ 0 & (x \leq 0) \end{cases} \tag{2.5}$$

Relu **関数**

$$y = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases} \tag{2.6}$$

各関数の x がノードの入力値，y がノードの出力値を示す．入力ベクトル x^l の各要素に対して活性化関数を適用する計算を $f(x^l)$ と表現するとき，式 2.2 と組み合わせることで，j 層の出力ベクトル y_j^l に関して以下の漸化式を得られる．

$$y_j^l = f(W_j y_{j-1}^l + b^l) \tag{2.7}$$

2.1.2　RNN と LSTM

RNN は Recurrent Neural Network を略した言葉であり，日本では再帰型ニューラルネットワークと呼ばれる．RNN は連続的な情報を扱うことができるニューラルネットワークモデルである．連続的な情報とは,「ある時点のデータが，それ以降のデータに何らかの影響を及ぼしている」と考えられる順序だったデータの集合である．例えば，株価や気温などのデータや，

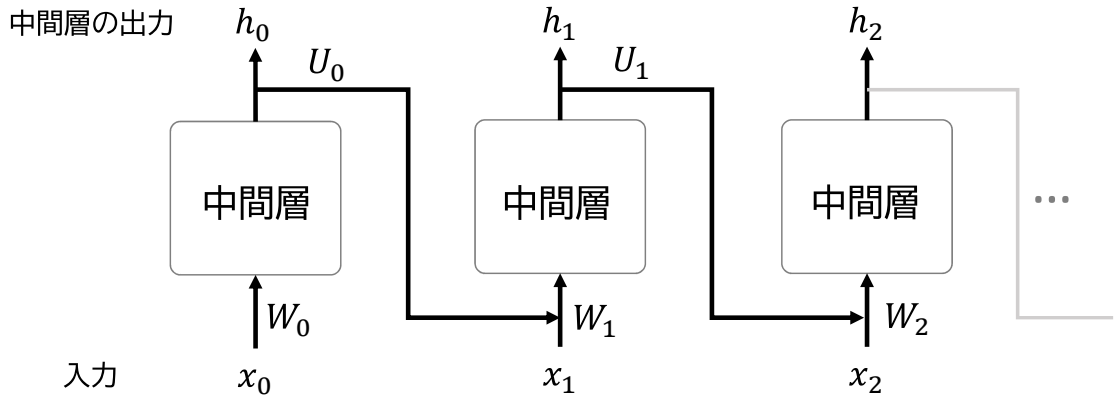


図 2.3. RNN レイヤーの概要

文章データなどがこれにあたる．これらをまとめて時系列データと呼ぶことにする．RNN はあるデータの計算結果を次のデータの計算にも利用することで，従来のニューラルネットワークでは難しかった，時系列データを用いた予測を可能にする．

図 2.3 に RNN レイヤーの概要を示す．RNN レイヤーでは，あるデータの中間層の出力ベクトルを，次のデータの中間層の入力ベクトルの計算に用いる．t 番目の中間層の出力ベクトル h_t は以下の計算式で計算できる．

$$h_t = f(W_t x_t + U_{t-1} h_{t-1} + b) \tag{2.8}$$

ここで，t 番目のデータの入力ベクトルを x_t，それに対する重み行列を W_t，t 番目の中間層の出力ベクトルに対する次のデータへの計算の重み行列を U_t，中間層のバイアスのベクトルを b ，用いる活性化関数を f と表す．

RNN は時系列データが多くなってしまうと，適切に学習できないという問題が生じる．RNN は次節で述べる学習の逆伝搬のとき，時系列データを遡って勾配の計算と重みの更新を行うが，層を通るごとに勾配が縮小しやすい．そのため，データ数が多すぎると，勾配が非常に小さくなり重みの更新がほとんど行われない勾配消失 [2] が起こってしまう．また，一部のケースでは，数値計算の不安定性による勾配の発散や不適切な重みの初期化によって勾配爆発が起こることがある．

これらの問題点を解決するために，従来の RNN を改良したのが LSTM [1] である．LSTM では，短期記憶の出力ベクトル h だけでなく，長期記憶の情報を保持するベクトル c を状態変数として用いることで，情報の長期記憶を可能にし，勾配消失などの問題点を解決している．LSTM は RNN の中間層の代わりに，LSTM ブロックを用いる．図 2.4 に LSTM ブロックの概要を示す．LSTM ブロックは，内部の情報の流れを制御する忘却ゲート，入力ゲート，出力ゲートによって，長期記憶と短期記憶の更新を行う．図中の x_t は t 番目の LSTM ブロックへの入力ベクトル，h_t は t 番目の LSTM ブロックの短期記憶の出力ベクトル，c_t は t 番目の LSTM ブロックから次の LSTM ブロックに渡される長期記憶のベクトルを表す．また，

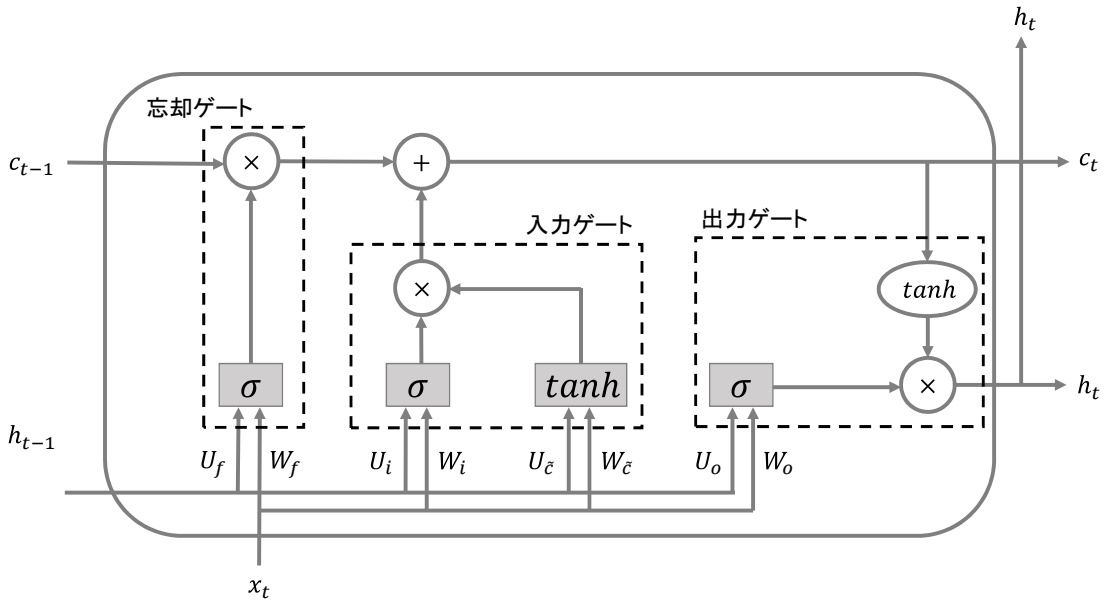


図 2.4. LSTM ブロックの概要

W_f と U_f は忘却ゲートの計算で用いる重み行列，W_i，U_i，$W_{\tilde{c}}$，$U_{\tilde{c}}$ は入力ゲートの計算で用いる重み行列，W_o と U_o は出力ゲートの計算で用いる重み行列，σ と $tanh$ はそれぞれ活性化関数として用いるシグモイド関数と tanh 関数を表す．

それぞれのゲートの役割と計算式を以下に示す．

忘却ゲート

長期記憶の古い情報をどれくらい忘れさせるかを制御する．長期記憶から廃棄する情報の割合 f_t を式 2.9 で計算する．b_f はこの計算で用いるバイアスのベクトルを表す．

$$f_t = \sigma(x_t W_f + h_{t-1} U_f + b_f) \tag{2.9}$$

入力ゲート

長期記憶に新しい情報をどれくらい覚えさせるかを制御する．長期記憶に貯める新情報 $\tilde{c}_t$ を式 2.10 で計算する．また，長期記憶に記憶させる新情報の割合 i_t を式 2.11 で計算する．$b_{\tilde{c}}$，b_i はこの計算で用いるバイアスのベクトルである．

$$\tilde{c}_t = tanh(x_t W_{\tilde{c}} + h_{t-1} U_{\tilde{c}} + b_{\tilde{c}}) \tag{2.10}$$

$$i_t = \sigma(x_t W_i + h_{t-1} U_i + b_i) \tag{2.11}$$

出力ゲート

長期記憶の情報をどれくらい抜き出して，短期記憶に含めるかを制御する．短期記憶に含める長期記憶の割合 o_t を式 2.12 で計算する．b_o はこの計算で用いるバイアスのベクトルである．

$$o_t = \sigma(x_t W_o + h_{t-1} U_o + b_o) \tag{2.12}$$

t 番目の LSTM ブロックにおいて，長期記憶のベクトル c_t は，以下の式によって更新される．

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{2.13}$$

また，短期記憶の出力ベクトルは，以下の式で更新される．

$$h_t = o_t \odot tanh(c_t) \tag{2.14}$$

2.1.3 機械学習モデルの学習

機械学習モデルは事前学習によってノード間の重みを最適化することで，答えが未知な入力データに対しても，目標以上の精度で予測を行うモデルを目指す．機械学習の学習手法は，大きく以下の 3 つに分けられる．

教師あり学習

入力データとそれに対応する正解ラベル (既知の正しい出力データ) のセットを学習データセットとして大量に準備し，入力データを用いたモデルの予測値が正解ラベルに近づくように，ノード間の重みを変化する．主に，各データが所属するクラスを推定する分類問題と，データの連続値を予測する回帰問題に用いられる．

教師なし学習

学習用の入力データには正解ラベルがついておらず，学習データそのものを何かしらの観点に基づいて，似ているデータ同士に分類する．代表的なアルゴリズムとして，主成分分析，クラスター分析，自己組織化マップ (SOM) などが挙げられる．

強化学習

教師あり学習や教師なし学習のように明確なデータをもとに学習を行うのではなく，プログラム自体が与えられた環境を観測し，目的として設定された報酬 (スコア) を最大化するための行動を学習する．囲碁 AI である AlphaGo [3] などのゲームの AI の学習手法として有名だが，最近ではロボット制御や自動運転などにも応用され始めている．

教師あり学習では主に誤差逆伝搬手法が用いられる．学習の流れは以下のようになる．

1. 各パラメータの初期値をランダムに決定
2. 入力データから予測値を算出
3. 予測値と正解ラベルから損失を計算
4. 損失を減少させるように重みを更新
5. 2 から 4 を繰り返す

2 の予測を行う処理は順伝搬と呼ばれる．ニューラルネットワークモデルでは式 2.7 を用いて入力層から出力層まで計算が行われる．RNN や LSTM などの時系列を扱うモデルでは，式 2.8 や 式 2.14 を用いて計算が行われる． 3 で損失を計算する関数を損失関数と呼ぶ．損失関数には二乗誤差，交差エントロピー誤差，平均絶対誤差などがある．4 の重みを更新する処理は逆伝搬と呼ばれる．順伝搬とは逆方向，すなわちニューラルネットワークモデルでは出力層

から入力層，RNN や LSTM などは新しいデータから古いデータに向かって重みの更新が行われる．損失関数を重みで微分して得られる勾配を用いて，損失が減少するように重みは更新される．この計算を行う手法を最適化アルゴリズムと呼ぶ．最も簡潔な最適化アルゴリズムである最急降下法の式は以下のようになる．

$$w_{t+1} = w_t - \alpha \nabla_w \mathcal{L}(w) \tag{2.15}$$

ここで，w_t は t 回目の更新後の重み，α は学習率，∇_w は重みでの微分，$\mathcal{L}(w)$ は損失関数を表す．

2.1.4　Skip-gram モデル

機械学習はコンピュータが理解できる数値表現しか扱えず，処理もすべて数値の演算で行われる．そのため，自然言語やプログラミング言語を処理する際に，入力単語をベクトル表現に変換することが一般的である．このとき，意味や文法的役割が似ている単語同士が似たベクトル表現を持つことで，汎化性能が高いモデルを学習することができる．

単語をベクトル表現に変える代表的な手法として，Mikolov らによって提案された Word2Vec [4] が挙げられる．この手法は，「ある単語の周辺の単語は関連性が高い単語である可能性が高い」という仮説に基づいている．Word2Vec は中心となる単語から前後を推測する Skip-gram モデル [4] と，前後の単語からその中心の単語を推測する CBOW モデル [4] の 2 つが存在する．この 2 つのモデルのうち，Skip-gram モデルの方がより優れた結果が得られる傾向にある [5].

Skip-gram モデルではまず，単純なニューラルネットワークを用いて，ある単語からその周辺の単語を予測するタスクの教師あり学習を行う．周辺の単語を前後何文字とするかを示す値をウィンドウサイズと呼ぶ．例として，データセットに以下の単語列を用いるとする．

```
There are ten fish swimming in the pond.
```

入力に用いる単語を `fish` だとした場合，ウィンドウサイズ 2 のときに `are`, `ten`, `swimming`, `in` を正解ラベルとして学習に用いる．入力単語と正解ラベルの数値変換には one-hot ベクトルを用いる．one-hot ベクトルとは，単語の種類の数だけ次元数を持ち， 0 と 1 しか値に持たないベクトルである．単語に対応した要素だけが 1 となり，他の要素はすべて 0 となる．例えば，データセットに上記の単語列だけを使用する場合，ピリオドを除いた単語の種類は 8 種類なので，各単語に対応した 8 次元の one-hot ベクトルを用いる．すなわち，`There` は [1,0,0,0,0,0,0,0]，`fish` は [0,0,0,1,0,0,0,0]，`pond` は [0,0,0,0,0,0,0,1] というように変換して，Skip-gram モデルの学習における入力と正解ラベルに用いる．

各単語に対応するベクトル表現は，学習したモデルの入力層から中間層への重み行列によって得られる．重み行列は，行数が入力層の次元数，すなわち one-hot ベクトルの次元数，列数が中間層の次元数である．ある単語の one-hot ベクトルと重み行列の行列積によって，その単語に対応するベクトル表現を得る．すなわち，ある単語の one-hot ベクトルの 1 の要素がの要素番号に対応する重み行列の行が，その単語に対応するベクトル表現になる．例えば，

There, are, ten, fish, swimming, in, the, pond の 8 種類の単語に対して学習を行った後の入力層から中間層への重み行列が以下になったとする．

$$\begin{bmatrix} 0.2 & 0.1 & 0.7 & 0.8 \\ 0.2 & 0.5 & 0.2 & 0.3 \\ 0.6 & 0.4 & 0.1 & 0.9 \\ 0.4 & 0.3 & 0.9 & 0.6 \\ 0.6 & 0.3 & 0.1 & 0.8 \\ 0.1 & 0.9 & 0.3 & 0.4 \\ 0.3 & 0.8 & 0.6 & 0.7 \\ 0.5 & 0.4 & 0.8 & 0.1 \end{bmatrix}$$

There の one-hot ベクトルが [1,0,0,0,0,0,0,0] だった場合，重み行列の 1 行目の [0.2, 0.1, 0.7, 0.8] が There に対応するベクトルになる．fish の one-hot ベクトルが [0,0,0,1,0,0,0,0] だった場合，重み行列の 4 行目の [0.4, 0.3, 0.9, 0.6] が fish に対応するベクトルになる．学習させるモデルの中間層のノード数を増減することで，単語に対応するベクトルの次元数を調整することができる，

2.1.5 字句解析によるトークン化

言語系の機械学習モデルでは，前節で説明した Skip-gram モデルなどを用いて各単語に対応したベクトルを事前に計算し，それを用いて学習と計算を行う．そのベクトルが単語同士の意味的な近さを表現するおかげで，文章の近似性などの分析を適切に行うことができる．しかしながら，自然言語の文章ではなくソースコードを機械学習モデルの入力に用いた場合，出現する可能性のあるすべての単語に対応するベクトルを事前に計算することは難しい．なぜならば，プログラミング言語では変数名などを自由に設定することができ，また使われる文字列や数値は様々なので，出現する単語の可能性が無限に存在するからである．

データ的に同じ単語を同一のトークンとして扱うことでこの問題を解決する．コードに出現しうる単語すべてに対応するようにトークンを定義することで，コードに出現する単語をすべて有限個の種類のトークンに変換することができる．コードの単語をすべてトークン化し，それらを Skip-gram モデルや機械学習モデルのデータセットに用いる．

また，トークン化することで，意味が近いコードは同一に扱うことができるという利点もある．例として，以下の 2 つの Python コードを考える．

```
i = "a" * 10

qxw37 = "qxw" * 37
```

単語単位でとらえると，この 2 つのコードは違うコードである．しかし，識別子を NAME，= を EQUAL，文字列を STRING，* を STAR，数値を NUMBER というトークンに変換すると，上記のコードはどちらも以下のトークン列に変換でき，同一データ構造の入力コードとして扱うことができる．

```
NAME EQUAL STRING STAR NUMBER
```

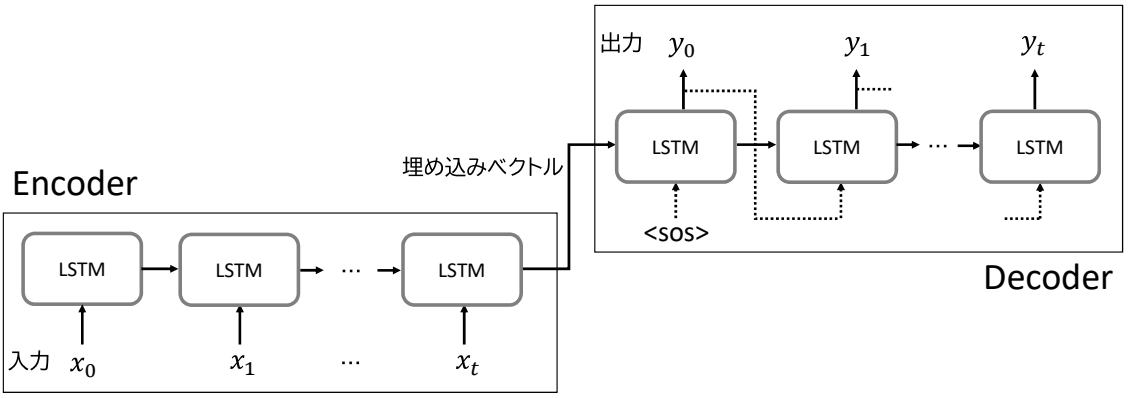


図 2.5. LSTM を用いた Encoder-Decoder モデル

コードをトークン化するには，一般的には字句解析器が用いられる．字句解析器はプログラミング言語では，主にコンパイル等のプログラム解析に使用され，言語それぞれの字句解析器が存在する．

2.1.6　Encoder-Decoder モデルと Transformer

Encoder-Decoder モデル [6] は時系列データを別の時系列データに変換するモデルである．モデルは Encoder と Decoder によって構成されており，まず Encoder が時系列データを埋め込みベクトルに変換し，Decoder がその埋め込みベクトルを別の時系列データに変換する．

図 2.5 に Encoder-Decoder モデルを LSTM を用いて構築したときの概要図を示す．Encoder では，LSTM ブロックを用いて入力データを時系列順に計算を行う．Encoder の最後の LSTM ブロックの出力が，埋め込みベクトルとして，Decoder に渡される．Decoder では，LSTM ブロックを用いて予測を行う．各 LSTM ブロックが出力した予測結果を時系列順に並べたものが，Encoder-Decoder モデルによって変換後の時系列データとなる．Decoder の LSTM ブロックへの入力は，一つ前の LSTM ブロックの予測結果を用いる．ただし，Decoder の最初の入力にはデコーダーが開始することを示す入力を行う．図の中の `<sos>` がこれを表す．Encoder-Decoder モデルを構築するモデルは，時系列処理ができるモデルであれば，LSTM である必要はない．例えば，LSTM と同じゲート機構を持つ機械学習モデルである GRU [7] や RNN を用いても Encoder-Decoder モデルを構築できる．

Encoder-Decoder は非常に有用なモデルであるが，モデルが理解できる情報量に限界があり，長い文章や未知の単語を含んだ文章に対しては，推論の精度が低いという問題がある．これは，Encoder が生成する埋め込みベクトルが固定長であることに起因する．その問題を解決したのが，Attention という機構である．Attention は Decoder が予測を行う各ステップにおいて，Decoder の現在の状態と Encoder の状態から，各入力データがこの予測にどの程度関連があるかを計算する．これにより，その Decoder の予測に対する各入力データの重要

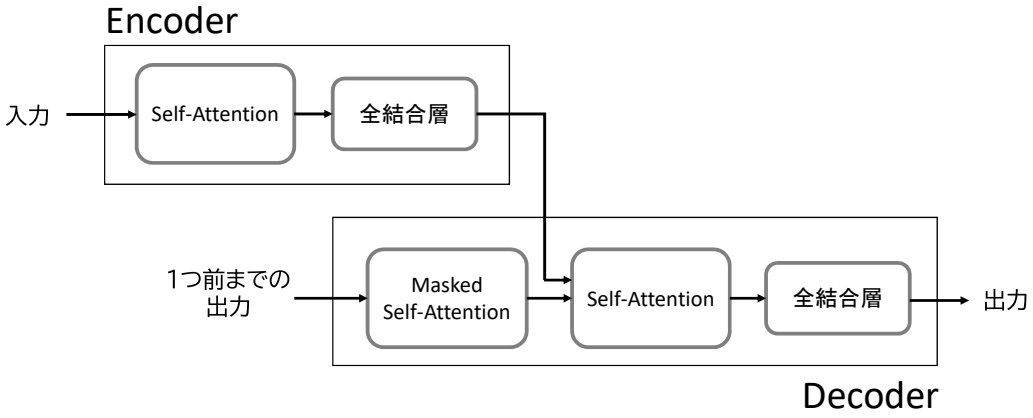


図 2.6. Transformer の概略図

度を考慮に入れて，Decoder は予測を行うことができる．この Attention 機構を応用したものとして，Multi-head Attention 機構がある．この機構は，従来の Attention 機構に比べて入力データの重要度を，より複数の観点でとらえることを可能にする．

Transformer [8] は， RNN や LSTM などの時系列データを処理する機械学習モデルを使用せずに，Multi-head Attention だけを用いて Encoder と Decoder を構成したモデルである．図 2.6 に，その概略図を示す．図の中の Self-Attention，および Masked Self-Attention の処理で，Multi-head Attention 機構が用いられる．また，全結合層にはニューラルネットワークが用いられる．Encoder の Self-Attention では，入力した時系列データのデータ同士の関係を数値化することで，時系列データの構造を理解する．Decoder の Self-Attention では，タスクに対するそれぞれのデータの重要度を計算する．

2.1.7 LLM

LLM は Large langage Models の略称であり，日本では大規模言語モデルと呼ばれる言語モデルである．LLM の大部分が Transformer をベースとしたモデルである．OpenAI 社の Kaplan らによると，自然言語モデルの性能と，「計算量」,「データ量」,「パラメータ数」の 3 つの要素との間にスケーリング則が成り立つと提唱している [9]．ここでいうスケーリング則とは，要素が増えることで予測の損失がべき乗則に従って減少することを指す．LLM の正式な定義はないが，特にこの 3 つの要素を大規模にした言語モデルを指すことが多い．

LLM はユーザーから入力プロンプトと呼ばれる自然言語での指示を受け取ることで，言語に関わる様々なタスクを行うことができる．LLM の行うことができるいくつかのタスクを以下に示す．

チャットボット

ユーザーの入力プロンプトに対して，人間のような返答をすることができる．また，

表 2.1. 主要な LLM

モデル	発表年月	パラメータ数	学習データ量	ソースコード
T5 [11]	2019 年 10 月	110 億	1 兆トークン	公開
GPT-3 [12]	2020 年 3 月	1750 億	3000 億トークン	非公開
LaMDA [13]	2022 年 1 月	1370 億	7680 億トークン	非公開
OPT [14]	2022 年 3 月	1750 億	1800 億トークン	公開
PaLM [15]	2022 年 4 月	5400 億	7800 億トークン	非公開
GPT-4 [16]	2023 年 3 月	非公開	非公開	非公開
LLaMA2 [17]	2023 年 3 月	700 億	2 兆トークン	公開

ユーザーの質問に対して，比較的正確な回答を行うことができる．

テキストに対する処理

LLM は自然言語のテキストに対する処理を非常に高精度で行うことができる．例えば，テキストの要約やテキストの別言語への翻訳，ユーザーの求める条件でのテキストの生成などを高いクオリティで行うことができる．

コードに対する処理

LLM は学習のデータセットに自然言語の文章だけでなく，プログラミングのソースコードが用いていることが多い．そのため，ユーザーが求めているコードの生成や入力されたコードの解説などを行うことができる．

LLM が従来の言語モデルに比べて飛躍的に性能を向上させ様々なタスクを行えるようになった大きな要因の 1 つは，文章全体の文脈の理解能力の向上によるものである．2018 年に Google 社の Devlin らが発表した BERT [10] は，複数箇所がマスクになっている文章において前後の文脈からマスクを予測する学習と，渡された 2 つの文章が連続した文章かどうかを判定する学習の 2 つの学習を行うことで，文章の文脈理解能力の高いモデルを開発した．BERT は 2018 年当時としては，質疑応答や翻訳などの自然言語処理タスクにおいて最高スコアを記録した．その後，現在に至るまでにより大規模でより性能の高い LLM が開発され続けている．表 2.1 にいくつかの主要な LLM の発表年月，パラメータ数，学習データ量，ソースコードの公開・非公開を示す．

一部の LLM はソースコードを公開しているため，ダウンロードすることでユーザーの構築した環境で実行することができる．しかし，LLM は莫大な数のパラメータを持ち，モデルサイズが非常に大きい．そのため，実行できる環境は限られてしまう．そこで，LLM を用いたシステムを開発するとき，LLM の API を利用することが一般的である．近年では，LLM を開発している主要な企業は新しいモデルの API を公開していることが多い．表 2.2 に 2024 年 1 月現在， LLM の API を公開している企業とそのモデルの一部を紹介する．

表 2.2. LLM API を公開している企業とモデル名

企業	モデル名
OpenAI/Azure	GPT-4 Turbo
OpenAI/Azure	GPT-3.5 Turbo
Anthropic	Claude 2.1
Meta	Llama 2 70b
Google	PaLM 2
Google	Gemini Pro

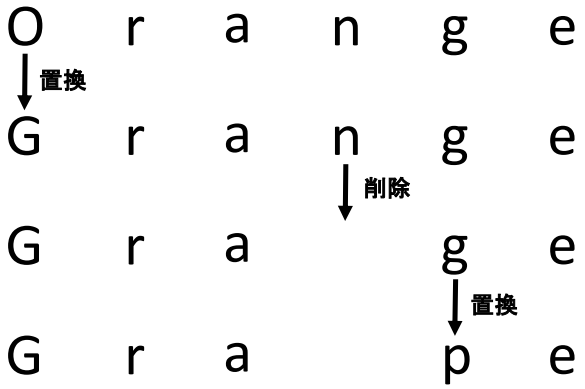


図 2.7. Orange から Grape に変換するときの操作

2.1.8 レーベンシュタイン距離

レーベンシュタイン距離 [18] は 2 つの文字列がどの程度異なっているかを示す指標の 1 つである．レーベンシュタイン距離は 2 つの文字列が一致するための最小の操作回数を表す．ここでいう操作は，以下の 3 つを指す．

挿入

文字列に新しい 1 文字を挿入する操作である．例えば red を read に変換するには，a を挿入する操作が 1 回必要である．

削除

文字列の 1 文字を削除する操作である．例えば， tape を tap に変換するには，e を削除する操作が 1 回必要である．

置換

文字列の 1 文字を別の 1 文字に置き換える操作である．例えば， bag を bug に変換するには，a を u に置き換える操作が 1 回必要である．

Listing 2.1. レーベンシュタイン距離を求める Python のコード

```
1  string_1 = "Orange"
2  string_2 = "Grape"
3  n = len(string_1)
4  m = len(string_2)
5  DP = [[0 for i in range(n+1)] for j in range(m+1)]
6
7  for i in range(n+1):
8      DP[0][i] = i
9  for j in range(m+1):
10     DP[j][0] = j
11
12 for j in range(1,m+1):
13     for i in range(1,n+1):
14         if string_1[i-1] == string_2[j-1]:
15             DP[j][i] = DP[j-1][i-1]
16         else:
17             erase_1 = DP[j][i-1] + 1
18             erase_2 = DP[j-1][i] + 1
19             replace = DP[j-1][i-1] + 1
20             DP[j][i] = min(erase, insert, replace)
```

例として，Orange から Grape に変換するときの操作を図 2.7 に示す．図のように，O から G への置換，n の削除，g から p への置換という 3 段階の操作で Orange から Grape に変換できる．そのため，Orange と Grape のレーベンシュタイン距離は 3 になる．

一般的に，レーベンシュタイン距離を求めるためには動的計画法が用いられる．動的計画法は，対象の問題を複数の部分問題に分割して，部分問題の計算結果を記録しながら解いていくアルゴリズムである．レーベンシュタイン距離を求める動的計画法の Python のコードを 2.1 に示す．コード内の処理の詳細を以下に示す．

1 ，2 行目

レーベンシュタイン距離を求める 2 つの文字列 string_1 と string_2 を定義する．

3 ，4 行目

それぞれの文字列の長さを求める．

5 行目

部分問題の計算結果を記録する配列 DP を定義している．DP の要素 DP[j][i] には，string_1 の i 文字までの文字列と，string_2 の j 文字目までの文字列のレーベンシュタイン距離を記録する．

7 から 10 行目

長さ 0 の文字列と長さ l の文字列のレーベンシュタイン距離が l であることに基づいて，配列 DP を初期化する．

12 から 20 行目

動的計画法によって DP のそれぞれの要素を計算する．14 行と 15 行目は string_1 の i 文字目と string_2 の j 文字目が同じ文字だった場合の計算である． 2 つの文字列の末尾に同じ文字を追加した場合，文字を追加する前の 2 つの文字列とレーベンシュタイン距離は変わらない．そのため，DP[j][i] = DP[j-1][i-1] となる．16 行目から 20 行目は string_1 の i 文字目と string_2 の j 文字目が違う文字場合の計算である．このとき，string_1 の i 文字目までの文字列と string_2 の j 文字目までの文字列のレーベンシュタイン距離の候補となる操作回数は以下の 3 つである．

- string_1 の i 文字目までの文字列と string_2 の j 文字目までまでの文字列が一致するには，string_1 の i 文字目までの文字列の末尾の文字を削除する操作を行った後に，string_1 の i-1 文字目までの文字列と string_2 の j 文字目までの文字列を一致させれば良い．つまり，操作回数は DP[j][i-1] + 1 である．
- string_1 の i 文字目までの文字列と string_2 の j 文字目までまでの文字列が一致するには，string_2 の j 文字目までの文字列の末尾の文字を削除する操作を行った後に，string_1 の i 文字目までの文字列と string_2 の j-1 文字目までの文字列を一致させれば良い．つまり，操作回数は DP[j-1][i] + 1 である．
- string_1 の i 文字目と string_2 の j 文字目 は 置換の操作を 1 回行うことで同じ文字に変えることができる．すなわち，置換の操作を 1 回加えるだけで， 2 つの文字列の末尾に同じ文字を追加した場合と同じになる．つまり，操作回数は DP[j-1][i-1] + 1 である．

この 3 つの操作回数のうち，最小の操作回数が string_1 の i 文字目までの文字列と string_2 の j 文字目までの文字列のレーベンシュタイン距離になる．

プログラム 2.1 を実行したとき，配列 DP は図 2.8 のように値の更新を行われる．図中の緑の枠の中が配列 DP を表し，黄土色のマスは初期化のときに更新された値である．また，末尾に同じ文字を追加した場合の値の更新を赤色の矢印，末尾に違う文字を追加した場合の値の更新を青色の矢印で表す．string_1 の i 文字目までの文字列と string_2 の j 文字目までの文字列のレーベンシュタイン距離は DP[j][i] の値なので，Orange と Grape のレーベンシュタイン距離は DP[5][6]，すなわち 3 となる．

2.2 複数言語を使い分ける開発環境における混乱

近年，プログラミングで扱う問題領域が広がったが，問題領域ごとに主流のプログラミング言語が異なるため，プログラミングの学習やプログラミングを用いた開発において，一人のプログラマが複数のプログラミング言語を使い分ける状況がしばしば見られる．例えば，大学の授業ごとに扱っているプログラミング言語が異なり，受講している学生が授業に応じて使用言

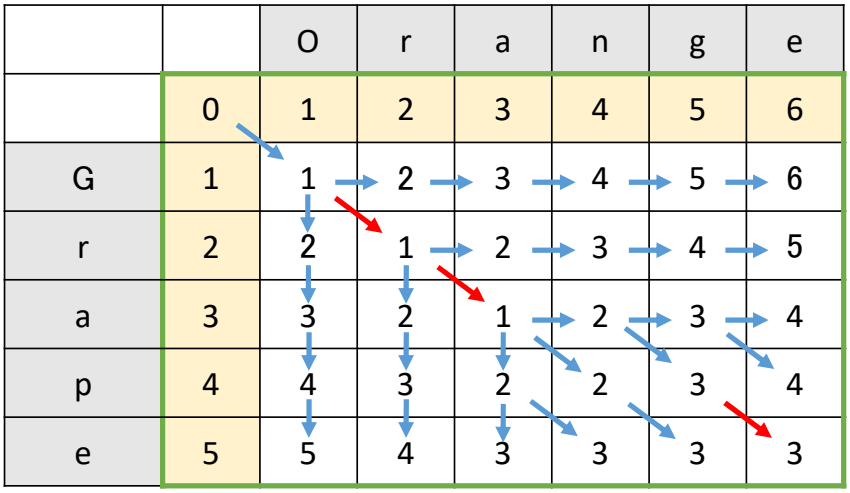


図 2.8. プログラム 2.1 を実行したときの配列 DP の値の更新

語を使い分けることがある．他にも，1 つのプロジェクトで複数の言語を使う場合がある．例えば，画像認識を行うスマホアプリの開発ではしばしば，画像認識部分は Python で，ユーザーインターフェース部分は Swift や Kotlin などの言語で，と複数の言語を使い分けて開発される．また，複数の開発プロジェクトに参加しているがプロジェクトごとに使用する言語が異なるという状況も考えられる．例えば，プログラマの人数が少ない組織で，JavaScript や HTML/CSS を用いて Web 開発をし，VBA や GAS を用いて業務効率化の開発をする，ということを一人のプログラマが任されることがある．

複数の言語を使い分ける開発環境では，開発者が混乱して，間違った言語の慣用句を書いてしまうことがある．ここでいう慣用句とは，典型的な処理のためのプログラムの書き方全般のことである．例えば，制御構文の書き方やオブジェクトの定義・処理の表現の仕方，主要関数の書き方などが慣用句である．プログラミング言語が異なると，同じ処理をする場合でも慣用句が異なることが多い．例として，配列の要素をすべて標準出力する同様の処理を Python，C 言語，JavaScript で書いたコードを図 2.9 に示す．この例だけでも，以下の慣用句の違いを確認することができる．

- C 言語と JavaScript は文末にセミコロンをつけているが，Python ではつけない．
- C 言語と JavaScript は波括弧で囲うことで，ブロックを表現するが，Python ではコロンとインデントでブロックを表現する．
- Python や JavaScript では配列の定義を角括弧を用いているのに対して，C 言語では波括弧を用いている．また，Python や JavaScript では型の定義を行っていないのに対して，C 言語では型を定義している．
- Python と C 言語では繰り返し処理に `for` 文を用いているが，その文法が異なる．JavaScript は `for` 文を用いずに高階関数である `foreach` メソッドを使用している．

Python

```python
array = ["a", "b", "c"]
for element in array:
    print(element)
```

C 言語

```c
const char* array[] = {'a', 'b', 'c'};
int array_count = sizeof(array) / sizeof(array[0]);
for(int i=0; i<array_count; i++){
    printf(array[i]);
}
```

JavaScript

```javascript
const array = ['a', 'b', 'c'];
array.forEach((element) =>
               console.log(element));
```

図 2.9. 配列の要素をすべて標準出力する処理の各言語による書き方

- 標準出力のときに，Python では `print` 関数，C 言語では `printf` 関数，JavaScript では `console.log` 関数を用いている．

この言語間の慣用句の違いによって，開発者が混乱して，間違った言語の慣用句を書いてしまうことがある．プログラミングに慣れていない人であれば，簡単な制御構造やよく使う関数名などでさえ混乱する可能性がある．すなわち，Python の `for` 文

```
for i in range(10)
```

と，C 言語の `for` 文

```
for(int i=0; i<10; i++)
```

を混乱することなどが考えられる．他にも，Python の標準出力の関数名 `print` を C 言語風に `printf` と書いてしまうことなどが考えられる．プログラミングに慣れている人でも，複雑な慣用句を他の言語でどう書けば良いかわからなることが考えられる．例えば，Java の Stream-API を用いて配列の奇数を 2 倍にして取得するプログラム 2.2 や，C# の LINQ のクエリ構文で書いた 2 つのリストを結合するプログラム 2.3，Ruby の `Enumerable` メソッドを用いてネストされた配列の各要素を 2 倍して平坦なリストに変換するプログラム 2.4 は，プログラミングに慣れている人でも，他の言語での書き方をすぐに思い出すのは難しい．

Listing 2.2. Java の Stream-API を用いて配列の奇数を 2 倍にして取得するコードの例

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
List<Integer> result = numbers.stream()
                            .filter(n -> n % 2 != 0)
                            .map(n -> n * 2)
                            .collect(Collectors.toList())
```

Listing 2.3. C# の LINQ を用いて 2 つのリストを結合するコードの例

```
var students = new List<Student>
{
    new Student { Id = 1, Name = "Alice" },
    new Student { Id = 2, Name = "Bob" }
};

var scores = new List<Score>
{
    new Score { StudentId = 1, Value = 90 },
    new Score { StudentId = 2, Value = 85 }
};

var query = from student in students
            join score in scores on student.Id equals score.StudentId
            select new { student.Name, score.Value };
```

Listing 2.4. Ruby の Enumerable メソッドを用いてネストされた配列の各要素を 2 倍して平坦なリストに変換するコードの例

```
nested_array = [[1, 2], [3, 4], [5, 6]]
flat_mapped = nested_array.flat_map { |sub_array| sub_array.map { |
    num| num * 2 } }
puts flat_mapped.inspect
```

2.3　異言語慣用句の修正候補を提示する入力支援

前節の問題意識に基づき，ユーザーが異言語の慣用句を書いてしまった場合に，それを簡単に修正できる入力支援を開発したい．本節では，目標とする入力支援のシステムの詳細と，それを開発する際の課題について説明する．

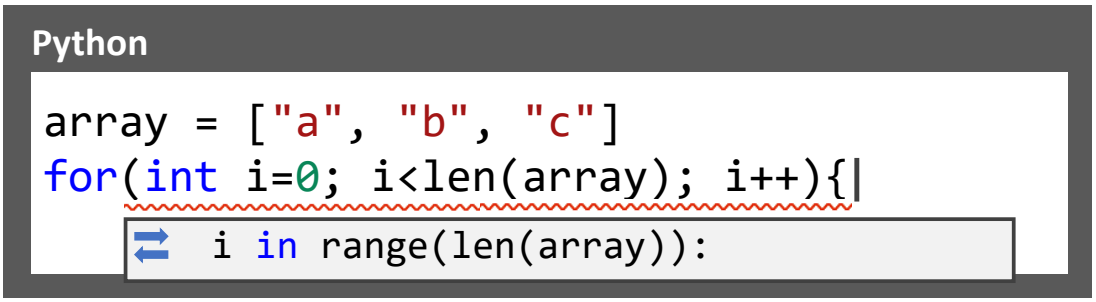


図 2.10. 目標とする入力支援のイメージ

2.3.1 目標とする入力支援

本研究が目標とする入力支援は，ユーザーが異言語の慣用句を書いてしまった場合に，それを即座に今書いている正しい言語に修正した置換候補をユーザーに提示するようなシステムである．この入力支援によって，うっかり異言語で慣用句を書いてしまった場合にそれを修正することができる．また，それだけではなく，今書いている言語での慣用句を忘れてしまった場合に，ユーザーが知っている言語でその慣用句を書くことで，書きたかった慣用句を補完することができる．図 2.10 に目標とする入力支援のイメージを示す．この図のように，Python を書いているときにユーザーが C 言語の `for` 文を誤って書いてしまったら，エディタが自動的にそれを検出して，正しく Python の `for` 文に直すための置換候補を提示できれば良い．

この入力支援の実現のために，以下のことが求められる．

- ユーザーがコード編集中に異言語で書かれた慣用句を書いたときに，異言語で書かれた慣用句の部分を今書いている言語に修正したい．そのため，今書いている言語と異言語が混在したコードに対して，コード翻訳が行えるモデルが必要である．
- コード補完の機能と同様に，ユーザーが異言語の慣用句を書いてしまったら，即座に翻訳と修正候補の提示を行いたい．そのため，ユーザーが書いているコードを随時分析する必要がある．
- 元のコードと翻訳したコードのうち，慣用句のコード翻訳を行った箇所だけをユーザーに提示したい．そのため，元コード内の異言語で書かれた慣用句の範囲とそれに対応する修正コードの範囲を特定する必要がある．

2.3.2 入力支援の開発の課題

2.1.7 節で説明したとおり，LLM は自然言語処理だけでなく，コード処理の面でも優れた性能を発揮できる．これは，コード翻訳のタスクにおいても例外ではない．例えば，GPT-3.5 モデルの入力プロンプトに翻訳するコードとともに以下の指示文を含める．

Listing 2.5. Python と C 言語が混在したコード

```
1 sum = 10
2 for(int i=0; i<sum; i++):
3     print(i)
```

Listing 2.6. プログラム 2.5 を GPT-3.5 を用いて Python に翻訳したときの出力

```
1 sum = 10
2 for i in range(sum):
3     print(i)
```

```
Translate this code to Python. Don't write any natural language.
```

C 言語の `for` 文を翻訳の元となるコードとして用いた場合,

```
for(int i=0; i<10; i++)
```

Python に翻訳された `for` 文を得られる.

```
for i in range(10):
```

また，Python と C 言語が混在したプログラム 2.5 を翻訳の元となるコードとして用いた場合，コード中の C 言語で書かれた部分だけが Python に翻訳されたプログラム 2.6 を得られる.

このように LLM は非常に精度の高いコード翻訳を行うことができ， それは複数言語が混在したコードに対しても当てはまる．すなわち，今書いている言語と異言語が混在したコードをすべて今書いているコードに修正することができる．そこで，LLM を用いれば，目標とする入力支援が簡単に実現できるように思えるが，実際には以下に示す 2 つの課題が存在する.

- 大量リクエストによるサーバー負荷とコスト増大
- 元コードと翻訳コードの慣用句部分の特定困難性

この 2 つの課題について説明する.

大量リクエストによるサーバー負荷とコスト増大

LLM を入力支援で用いる場合は LLM の API を用いることが望ましい．LLM は非常にサイズの大きいモデルであり，これから更にサイズの大きい LLM が開発されることが予想される．そのため，オープンソースの LLM をダウンロードして使用する場合，最新の LLM を永続的に実行できる環境を用意するのは難しい． LLM の API を用いれば，開発元のサーバーから LLM を使用することができるので，モデルを置く環境を構築する必要がない．また，LLM の中にはソースコードを公開していないクローズドなものも多いため，目的のモデルをそもそもダウンロードできない可能がある．これらのクローズドなモデルも，最新のモデ

Listing 2.7. LLM に送る入力プロンプト

```
Translate this code to Python. Don't write any natural language.
if order_1 = "add" and order_2 = "all": #Check the order
  int_list = list(map(int, str_list))#Change string type to int type
  sum = 0 #Define variable to record the sum
  for (int i=0; i<len(int_list); i++):
    sum += int_list[i]
```

Listing 2.8. 入力プロンプト 2.7 に対する出力

```
if order_1 = "add" and order_2 = "all": #Check the order
  int_list = list(map(int, str_list))#Change string type to int type
  sum = 0 #Define variable to record the sum
  for i in range(len(int_list)):
    sum += int_list[i]
```

表 2.3. 主要 LLM に対して入力プロンプト 2.7 と 出力 2.8 の API 通信を 1000 回行ったときにかかるコスト

モデル	入力プロンプトに対するコスト	出力に対するコスト	合計のコスト
GPT-4 Turbo[*1]	$0.72	$1.65	$2.37
GPT-3.5 Turbo[*2]	$0.072	$0.11	$0.183
Claude 2.1[*2]	$0.576	$1.32	$1.896
Llama 2 70b[*3]	$0.0468	$0.15125	$0.19805
PaLM 2[*4]	$0.07275	$0.11	$ 0.18275
Gemini Pro[*5]	$0.07275	$0.11	$ 0.18275

ルは API が公開されることが多い.

ただし，LLM の API を用いてこの入力支援を開発する場合，LLM のサーバーに大量のリクエストを送ってしまう問題が起こる．目標とする入力支援は，ユーザーが異言語の慣用句を書いてしまったら，即座に翻訳と修正候補の提示を行うために，ユーザーが書いているコードを随時分析することを想定している．そのため，ユーザーがコード編集を行っている間に何度もモデルを呼び出すことになる．例えば，コード 1 行書くときに平均 10 回分析を行う場合，100 行のコード編集を行うときに，1000 回モデルを呼び出すことになる．入力支援を使用するユーザーが増加した場合，それらのユーザーが同時にコード編集を行うと，LLM のサーバーに短時間のうちに大量のリクエストを送り，サーバーに負荷をかけしまうことが考えられる．

Listing 2.9. LLM に送る入力プロンプト

```
1 Translate this code to Python. Don't write any natural language.
2 for i in range(20):
3     sum_val+=math.sqrt(i)
4     product*=math.sqrt(i)
5 printf(sum_val)
6 printf(product)
```

また，LLM のサーバーに大量のリクエストを送った場合，コストの面でも問題が生じる．例えば，LLM にコード翻訳を行う入力プロンプト 2.7 を送り，それに対して 出力 2.8 を得られたとする．LLM の API のコストのほとんどは，トークン数もしくは文字数によって計算される．現在のトークン数の算出方法だと，2.7 の入力プロンプトは 72 トークン，291 文字，2.8 の出力は 55 トークン，220 文字として計算される．主要な LLM の API に対して，この入力プロンプトと出力の API 通信を 1000 回行ったときにかかるコストを表 2.3 に示す．例えば，コード の分析を行うためにコード 1 行あたり平均 10 回モデルと API 通信を行う場合，100 行ほどのコード編集の入力支援を行うだけで，これだけのコストがかかってしまうことになる． これは，不特定多数のユーザーに，コード編集中に常に使用される入力支援としては，好ましくない．

元のコードと翻訳した後のコードの慣用句部分の特定困難性

LLM のコード翻訳によって今書いている言語と異言語が混在したコードをすべて今書いている言語に修正することができるが，具体的にコードのどこの部分を修正したかの情報を LLM から得るのは難しい．現在の LLM の性能では，コードの修正した箇所も出力するように指示した場合，自然言語を用いて修正した箇所の説明が返ってくることが多いため，機械的に修正箇所を特定することは困難である．自然言語の使用を禁止する指示を含めた場合においても，修正したコードの具体的な場所を明示的に得られる入力プロンプトは我々の知る限りではない．

また，目標とする入力支援では，ユーザーが書いているコード全体を分析するのではなく，カーソル付近のコード数行をモデルに渡してコード翻訳を行うことを想定している．その場合，モデルに渡すコード数行の中で使用される変数やライブラリが，そのコード内で定義されていない可能性がある．LLM のコード翻訳では，この変数やライブラリの定義文を勝手に補完してしまう．また，LLM の出力のコードの形式はフォーマット化されていること多く，改行や空白を勝手に加えてしまうことがある．実際に，コード翻訳を行う入力プロンプト 2.9 を

*1 https://openai.com/pricing
*2 https://www-files.anthropic.com/production/images/model_pricing_dec2023.pdf
*3 https://replicate.com/pricing
*4 https://cloud.google.com/vertex-ai/docs/generative-ai/pricing
*5 https://ai.google.dev/pricing

Listing 2.10. 入力プロンプト 2.9 に対する出力

```
1 import math
2
3 sum_val = 0
4 product = 1
5 for i in range(20):
6     sum_val += math.sqrt(i)
7     product *= math.sqrt(i)
8
9 print(sum_val)
10 print(product)
```

GPT-3.5 モデルに送ったときに，出力 2.10 が得られた．このコード翻訳によって，C 言語の printf 関数で書かれた標準出力が python の print に修正されている．しかし，それとは別に，渡した入力プロンプトのコード内で定義されていない変数 sum_val や product，ライブラリ math の定義文が補完されていることがわかる．また，入力プロンプトのコード内にはなかった空白行が 2 行追加されており，for 文の中で sum_val と product の演算を行っている 2 行で演算子の左右に空白が足されていることがわかる．

本研究が目標とする入力支援は，異言語で書いてしまった慣用句だけを修正するものであり，それ以外の箇所ではユーザーの書いたコードに修正を加えないことを理想としている．異言語で書いてしまった慣用句の修正は，上記の例だと printf から print への修正がこれにあたる．そのため，変数やライブラリの定義文の補完や，改行や空白の追加などの修正箇所を含めずに，異言語で書かれた慣用句を今書いている言語に修正した箇所だけを特定してユーザーに提示したい．これは，LLM だけを用いた手法だと難しい．

2.4 関連研究

LLM や大規模機械学習モデルを用いたコード補完に関する研究や，それを活用した入力支援システムは既に存在する．また，LLM の開発が進む以前にも，機械学習モデルによってコード翻訳を試みる研究が行われている．本節では，既存のコード翻訳やコード補完に関連する研究や入力支援システムを紹介し，本研究が目標とする入力支援との違いを述べる．

2.4.1 GitHub Copilot (2021)

Github Copilot [19] は，書きかけのコードやコメントからユーザーがどのようなコードを書きたいのかをモデルが予測して続きのコードを補完する，LLM を用いた入力支援ツールで

ある．この入力支援ツールは VS code [*6] や Visual Studio [*7]，Neovim [*8]，JetBrains IDEs [*9]といった主要なコードエディタや統合開発環境からも使用することができ，多くの人が利用している．

LLM には GitHub 社と OpenAI 社が共同で開発した Codex [20] というモデルが用いられている．Codex は GPT-3 [12] をベースにして数十億行の公開されたソースコードでトレーニングされたおり，自然言語に関する文脈理解と生成に優れていた GPT-3 に比べて，よりプログラミング言語に関する文脈理解と生成に優れたモデルである [21].

GitHub Copilot のコード補完は，入力済みのコードやコメントから続きのコードを予測するものである．そのため，入力済みのコードを正しいコードに修正することを目標とする本研究の入力支援のコード補完とは異なる．

2.4.2　Ciniselli (2021)

Ciniselli らは，BERT をベースとしたモデル RoBERTa [22] に対して，コード補完の限界と能力を探る実験を行った [23]．主に，以下に示す 3 つの要素がコード補完の性能に与える影響を検証した．

粒度

トークン単位，for 文の完全な条件定義や関数呼び出しの引数定義などの構成要素単位，波括弧で囲まれたブロック単位という異なる粒度のコード片に対するコード補完の精度を比較した．また，それぞれの粒度においても，補完するトークンの数を変えての検証も行った．

特異性

様々な用途で書かれた Java のソースコードを含むデータセットと，Android で用いられている Java のソースコードのデータセットによってそれぞれ学習されたモデルに対して，そのデータセットに対するコード補完の精度を比較した．

抽象化

コード中の変数名の表記の仕方を統一したコードのデータセットと，何も変更を加えていないソースコードのデータセットによってそれぞれ学習されたモデルに対して，そのデータセットに対するコード補完の精度を比較した．

結果として，粒度が細かいほど，そして補完するトークンが少ないほど精度が高くなることが確認された．また，特殊なデータセット及びコードの抽象化を用いたモデルの方が，より良い精度を示した．

ただし，この研究も GitHub Copilot と同様に入力済みの数語からコードの残りを予測する

[*6] https://code.visualstudio.com/
[*7] https://azure.microsoft.com/ja-jp/products/visual-studio
[*8] https://neovim.io/
[*9] https://www.jetbrains.com/ja-jp/

コード補完に対するものであるため，入力済みのコードを正しいコードに修正することを目標とする本研究の入力支援のコード補完とは異なる．

2.4.3 Roziere (2020)

Roziere らは，Attention [8] 付きの Encoder-Decoder モデル [6] を用いて，プログラミング言語間の関数単位でのコード翻訳を可能にする TransCoder を開発した [24]．TransCoder は Lample らによって提唱された教師なし学習による機械翻訳の 3 つの原則，初期化，言語モデリング，逆翻訳を用いて学習された [25]．また，モデルの学習および検証は，Python, Java, C++ の 3 つの言語間の関数単位の翻訳を対象に行われた．

TransCoder は，Java と C++ の間では，Java から C++ への翻訳を 80.9%，C++ から Java への翻訳を 60.6% 成功させた．しかし，Python と Java の間で成功率は，Python から Java への翻訳が 24.7%，Java から Python への翻訳が 35.5% と比較的低い数値になった．

TransCoder は，関数単位の比較的大きな粒度のコード片を翻訳対象としたモデルである．また，TransCoder に入力として与える翻訳の元となるコードは 1 つの言語で書かれていることが前提となっている．そのため，複数の言語が混在したコードの中で，慣用句単位の翻訳を行う本研究のコード翻訳の処理には利用できない．

第 3 章

LLM を応用した入力支援システムの提案

我々は，コード編集中のユーザーのカーソル付近のコードに異言語の慣用句があった場合に，その慣用句を今書いている言語に翻訳したものを修正候補としてユーザーに提示する入力支援システムの開発を目標としている．本論文では，LLM のコード翻訳に，異言語の慣用句の混入を判定する処理と，元のコードと翻訳後のコードから異言語の慣用句部分だけを抜き出す処理を組み合わせることで，このシステムを開発する手法を提案する．

本章では，提案する入力支援システムの概要と，それぞれの処理の手順について述べる．まず，入力支援システム全体の概要と，異言語の慣用句の混入を判定する処理および元のコードと翻訳後のコードから異言語の慣用句部分だけを抜き出す処理にどのような役割があるかを 3.1 節で説明する．また，3.2 節ではシステムの前半部分にあたる，異言語慣用句の混入を判定する処理について，3.3 節ではシステムの後半部分にあたるコード翻訳と元のコードと翻訳後のコードから異言語の慣用句部分を抽出する処理について具体的な処理の手順を説明する．

3.1　提案する入力支援システムの概要

本節では，LLM のコード翻訳に，異言語の慣用句の混入を判定する処理と，元のコードと翻訳後のコードから慣用句を抽出する処理を組み合わせた入力支援システムの概要と，2.3.2 節で述べた課題へどのように対処したのかを述べる．

図 3.1 に提案する入力支援システムの概要を示す．まず，カーソル付近の数行のコードを取得し，そのコードに対して異言語の慣用句の混入を判定する処理を行う．この処理で「異言語の慣用句が混入していない」と判定された場合は，次の処理に進まない．「異言語の慣用句が混入している」と判定された場合は，コードを LLM に送り，コード翻訳を行う．その後，取得した元のコードと，そのコードを LLM によって翻訳した後のコードから慣用句部分を抽出する処理を行い，異言語の慣用句と今書いている言語に翻訳された慣用句を取得する．それらの情報を用いて，ユーザーに修正候補を提示する．

LLM のコード翻訳を行う前に，異言語の慣用句の混入を判定する処理を行うことで，コー

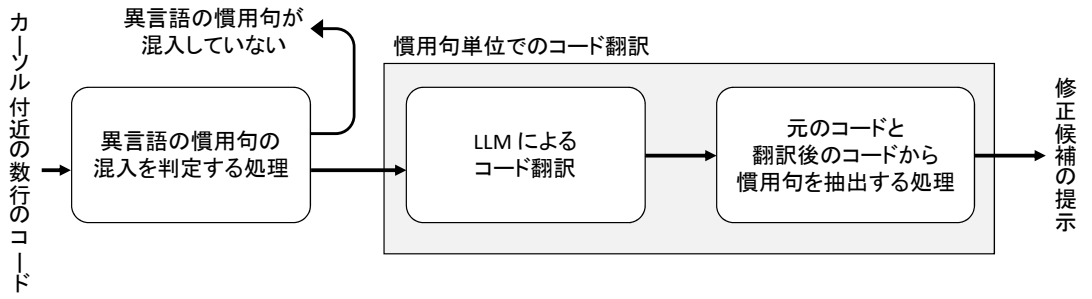


図 3.1. 提案する入力支援システム

ド翻訳を行う必要のあるコードかどうかを取捨選択でき，LLM へのリクエストの数を制限できる．それにより，サーバーへの負荷を軽減し，LLM の API にかかるコストも減少することができる．例えば，1000 回のコード分析において，異言語の慣用句の混入を判定する処理で「異言語の慣用句が混入している」という判定が 3 回しか出なかった場合は，LLM へのリクエスト数が 1000 分の 3 にすることができる．

判定を行う機械学習モデルには，LSTM を用いる．LSTM は LLM に比べて非常にサイズの小さく，自作のモデルを簡単に開発できる．そのモデルは使用するときに当然コストはかからない．また，LSTM は LLM の API 通信に比べて非常に応答時間が短い．そのため，異言語の慣用句が混入していない場合のシステムの応答が，LLM だけを用いている場合に比べて非常に早くなるという利点もある．

LLM によるコード翻訳と，元のコードと翻訳後のコードから慣用句を抽出する処理を組み合わせることで，慣用句単位でのコード翻訳を可能にする．2.3.2 節で述べたとおり，LLM のコード翻訳だけでは，翻訳の元になるコードのどの部分が異言語の慣用句にあたり，翻訳後のコードのどの部分がそれを翻訳した慣用句なのかを特定することが難しい．そこで，翻訳の元になるコードと翻訳後のコードを単語単位から共通部分を探し，その結果から元のコード内の異言語の慣用句の範囲，翻訳後のコード内の翻訳された慣用句の範囲を特定する処理によって，LLM が苦手とするタスクを補完する．また，翻訳された慣用句の中に LLM のコード翻訳による勝手な補完を含まないように，この処理を工夫する．

3.2 異言語の慣用句の混入を判定する手順

本手法では，まずユーザーから取得したコードに異言語の慣用句が混入しているか否かを判定する処理を行う．コード編集中のユーザーのカーソル付近数行のコードを受け取り，そのコードの中に異言語で書かれた箇所があるか否かを判定する．判定には，異言語である可能性を示す指標を用いる．本論文では，その指標を *diff* 値と表すことにする．また，*diff* 値を元に異言語の慣用句が混入しているか判定の判定に閾値を用いる．その閾値を 閾値 α と表すことにする．また，3.3.3 節で説明する修正箇所を処理に用いるため，*diff* 値を元に異言語で書

かれている可能性の高い部分も取得する．

コードを取得してから異言語の慣用句の混入を判定するまでの手順を以下に示す．

1. コードを字句解析し，トークン列に変換する．
2. トークン列を n-gram に分割する．
3. 各 n-gram に異言語が含まれているかどうかを LSTM を用いて判定する．
4. 3 の結果を用いて，各トークンに対して *diff* 値を計算する．
5. 最大の *diff* 値が閾値 α を超えているかどうかを判定する．

本節では，それぞれの手順の詳細を述べる．3.2.1 節では，コードのトークン化に用いた字句解析手法，3.2.2 節ではそのときに用いたトークンの種類について説明する．3.2.3 節では n-gram の分割，3.2.4 節では LSTM を用いた各 n-gram に対する異言語か否かの判定手法，3.2.5 節では *diff* 値の計算方法，3.2.6 節では異言語の慣用句が混入しているか否かの判定方法の詳細を述べる．

3.2.1　字句解析手法

2.1.5 節で説明したとおり，機械学習モデルを用いて異言語の慣用句の混入を判定する場合，まず字句解析をおこなってプログラムをトークン列に変換する必要がある．ただし，今回のコードのトークン化は，今ユーザーが書いているプログラミング言語の字句解析器を用いても，うまくいくとは限らない．その言語とは異なる言語が混入している可能性を踏まえて，字句解析できなければならないからである．

そこで我々は，本来の言語が何言語であっても Python の字句解析器を用い，それによる字句解析の結果にヒューリスティックな処理を加えることで，この問題に対処している．例えば本来の言語が C++ であり，プログラムの大半が C++ で書かれていても，Python の字句解析器を用いて解析をおこなう．Python の字句解析器を使うのは以下の理由からである．

- Python の 単行コメントの記号として使われる `#` は，C++ のディレクティブの接頭記号など，他言語での使用用途が限定的である．
- Python の複数行コメントの記号として使われる `"""` は，他の言語ではあまり使われない．
- C++ や Java の複数行コメントの記号として使われる `/*` は Python では使われない．
- Python は一重引用符と二重引用符のどちらでも文字列を囲める．C++ や Java では一重引用符で囲めるのは単一の文字なので，C++ や Java の字句解析器では一重引用符に囲まれた Python の文字列を解析するときにエラーになる可能性がある．

Python の字句解析器による解析結果に加えるヒューリスティックな処理は次のようなものである．例えば Python の字句解析器は次のような行を字句解析すると

```
#include <iostream>
```

`#` に続く部分を全てコメントトークンに変えてしまうので，`#` を接頭記号とするディレクティ

ブを含む上の行は次のトークンに変換される．

```
COMMENT
```

この行は本来は次のようなトークン列に変換されるのが望ましい．

```
HASH INCLUDE NAME
```

そこで，コメントトークンとしてトークン化されたものは，トークン化前の文字列の最初の部分を参照し，`include` や `define` といった文字列と一致した場合，`#` を除いた部分でトークン化し直す．

また，Python の字句解析器はプログラム中の `//` を整数除算の記号として認識してしまうため，`//` をコメントアウトの記号として使用する言語を字句解析した場合に，コメント部分がコメントトークンとしてトークン化されない．コメント部分もそれ以外の部分と同じようにトークン化されてしまう．例えば次の行は

```
// Written in C
```

次のようなトークン列に変換される．

```
DOUBLESLASH NAME NAME NAME
```

この行はコメントなので，次のようなトークン列に変換されるのが望ましい．

```
DOUBLESLASH COMMENT
```

これを実現するため，`//` から改行までのトークン列を分析し，それが整数除算としての `//` に続くものとして考えにくいトークン列だった場合，そのトークン列全体を 1 つのコメントトークンに変換する．

このようなヒューリスティックスでも機能するのは，字句解析器の結果得られるトークン列が機械学習モデルの入力として使われるだけだからである．Python のプログラムを書いているときに誤って C++ の慣用句を書いてしまったからといって，C++ の慣用句の部分を C++ の字句解析器と同様に解析する必要はない．適当なトークン列に分解することができ，その慣用句のプログラミング言語を機械学習モデルがそのトークン列から推定できればよい．

3.2.2 使用するトークンの種類

今回の手法では，字句解析器が通常使用するトークンの種類に加えて，以下のトークンを区別して使用する．

1. 予約語
2. 標準ライブラリの関数名やメソッド名
3. その他よく使用する識別子

3.2.1 節で述べた Python の字句解析器として，Python の標準モジュール tokenize [*1] のジェネレータ `tokenize.tokenize()` を使用する．この `tokenize` が生成する主なトークン

[*1] `https://docs.python.org/3/library/tokenize.html`

表 3.1. tokenize が生成するトークン

データの種類	トークンの表記
識別子	NAME
数値	NUMBER
文字列	STRING
演算子・記号	OP
コメント	COMMENT
改行	NL
字下げ	INDENT
字上げ	DEDENT

表 3.2. 演算子・記号のトークン

演算子・記号	トークンの表記	演算子・記号	トークンの表記
(	LPAR	!=	NOTEQUAL
)	RPAR	>=	LESSEQUAL
[	LSQB	<=	GREATEREQUAL
]	RSQB	<<	LEFTSHIFT
:	COLON	>>	RIGHTSHIFT
,	COMMA	**	DOUBLESTAR
;	SEMI	+=	PLUSEQUAL
+	PLUS	-=	MINEQUAL
-	MINUS	*=	STAREQUAL
*	STAR	/=	SLASHEQUAL
/	SLASH	%=	PERCENTEQUAL
>	LESS	<<=	LEFTSHIFTEQUAL
<	GREATER	>>=	RIGHTSHIFTEQUAL
=	EQUAL	**=	DOUBLESTAREQUAL
.	DOT	//	DOUBLESLASH
%	PERCENT	//=	DOUBLESLASHEQUAL
{	LBRACE	@	AT
}	RBRACE	@=	ATEQUAL
==	EQEQUAL		

を表 3.1 に示す．このうち，演算子・記号トークン OP は，tokenize.tokenize() が返す named tuple の exact_type プロパティをチェックすれば更に細かく分けられる．これらを表 3.2 に示す．

ユーザーが異言語の慣用句を書いてしまったかを判定するためには，tokenize が生成するトークンだけでは不十分である．なぜならば，言語の違いによる慣用句の差異が判別できないことがあるからである．例えば，tokenize が生成するトークンだけでは，以下の Python の標準出力文と

```
print("Hello world!")
```

C 言語の標準出力文が，

```
printf("Hello world!")
```

同じトークン列に変換されてしまう．

```
NAME LPAR STRING RPAR
```

他にも，全く違う機能を持つコードが似たトークン列になってしまうことが考えられる．例えば，Java のストリーム API を用いたフィルタリングのコードと

```
numbers.stream().filter(n -> n < max)
```

Python の numpy 型を list 型に変換して，特定の値の要素のインデックスを list の中から検索するコードが，

```
np_array.tolist().index(a, start, end)
```

似たトークン列に変換されることが考えられる．

```
NAME DOT NAME LPAR RPAR DOT NAME LPAR ...
```

この問題を解決するために，識別子のうち，各言語の予約語と標準ライブラリの関数名やメソッド名，その他よく使用する識別子は個別のトークンとして扱う．

予約語とは，プログラミング言語によってあらかじめ用途が決まっており，コード上で変数名や関数名などとして扱えない単語である．例えば，繰り返し処理を行うための for や 関数の呼び出し元へ値を返す return などがそれに当たる．標準ライブラリとは，そのプログラミング言語の実行環境で事前にインポートされているライブラリである．つまり，パッケージ管理システムや公開サイトなどからインストールしなくても，すぐに利用できる関数やメソッドがこれにあたる．言語ごとの標準ライブラリの関数名やメソッド名は，その言語の公式ホームページや公式ドキュメントなどから取得することができる．その他よく使用する識別子とは，予約語や標準ライブラリの関数名やメソッド名に当てはまらない識別子のうち，各言語で登場回数の多い識別子のことを指す．例えば Python でしばしば利用される外部ライブラリ NumPy をコード内で使用するときの略称 np などがこれにあたる．各言語に対して，特定の識別子がソースコードに登場する頻度を調べ，それぞれの上位 100 個ほどを個別のトークンとして扱うことにする．

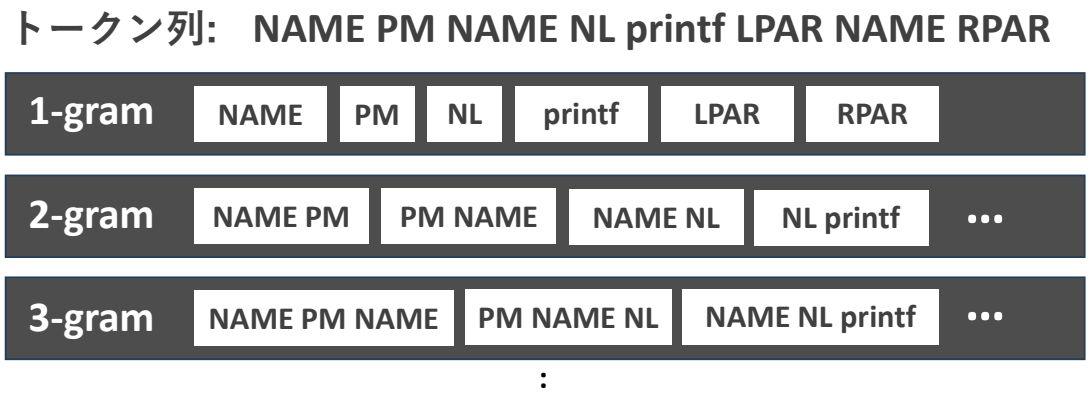


図 3.2. n-gram 分割の例

3.2.3　n-gram への分割

異言語の慣用句の混入を判定する次の手順として，LSTM の入力に用いるための様々な長さのトークン列を作成する．n-gram とは，何らかの列を連続する n 個の列の組みにした集合である．図 3.2 にトークン列を 1-gram から 3-gram までに分割した例を示す．本手法では，字句解析によってコードから変換されたトークン列をトークンを単位として，最大 20-gram に分割する．実際には，このトークン列の前後に偽のトークン列を加えてから，n-gram 分割を行う．その理由と詳細は 3.2.5 節で解説する．

様々な長さの n-gram を用いるのは，単語レベルの短い慣用句から，制御構造レベルの長い慣用句まで，機械学習モデルが判定できるようにするためである．長い n-gram だけを判定に用いた場合，例えば Python の標準出力の関数名 print が，C 言語の関数名 printf に変わっていた場合に，その間違いを認識できない可能性が考えられる．一方で，短い n-gram だけを判定に用いた場合，例えば繰り返し構文など，言語による差異が単語の順序や構造に表れる慣用句の間違いを認識できない可能性がある．そのため，1-gram から 20-gram まで様々な長さの n-gram に対して異言語で書かれているかの判定を行い，それぞれの判定結果を総合的に踏まえて，各トークンがどれくらい異言語である可能性があるかの計算を行う．その具体的な方法は 3.2.5 節で述べる．

3.2.4　LSTM を用いた判定

事前学習した LSTM を用いて，3.2.3 節で作成した n-gram それぞれに対して，異言語の部分が含まれているかどうかを判定する．この判定結果は，異言語である可能性を示す指標である *diff* 値を各トークンに対して計算するときに使用する．

図 3.3 は n-gram のトークン列に異言語が含まれているかどうかを判定するまでのモデル内部の計算の概要を示している．判定までの計算の流れを以下に示す．

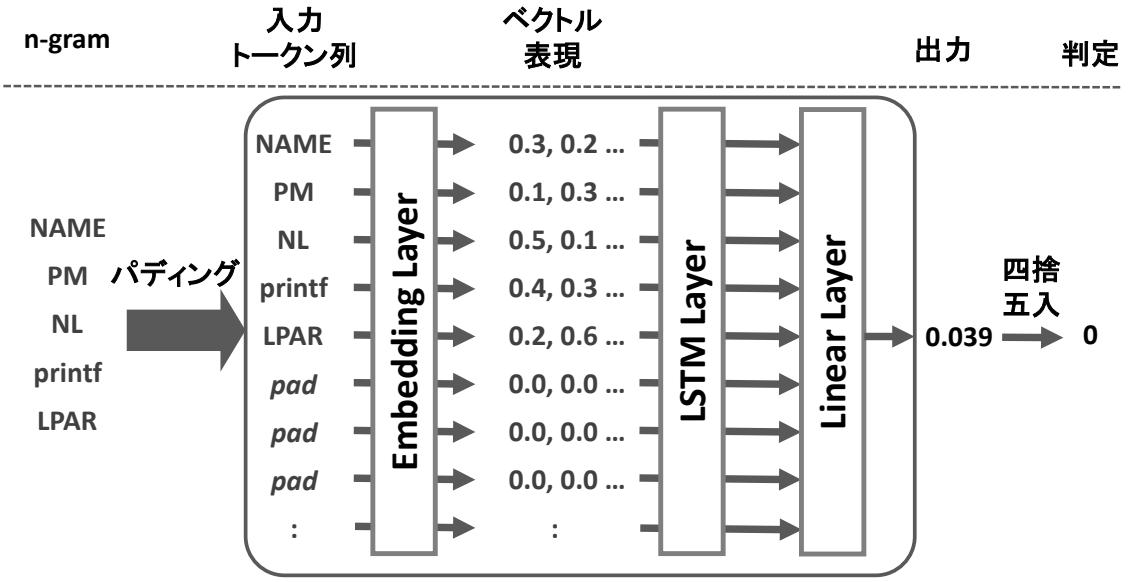


図 3.3. n-gram に異言語が含まれるか判定する LSTM モデルの概要

1. 判定する n-gram にパディングを行うことで，入力のトークン列を作成する．パディングとは，計算にあまり影響を与えないパディングトークンを n-gram の後ろに追加し，モデルが受け取る系列長を同一にする手法である．図 3.3 の *pad* がパディングトークンを示す．今回の手法では，使用する最も長い n-gram が 20-gram になるので，系列長の長さが 20 になるようにパディングを行う．
2. Embedding レイヤーで入力のトークン列をベクトル表現に変換する．このときのベクトル表現は，LSTM を学習させたときに使用したベクトル表現を使用する．
3. LSTM レイヤーを経て，Linear レイヤーで 1 次元の数値に変換する．
4. 得られた数値を四捨五入することで，0 または 1 を出力する．出力が 0 の場合，n-gram に異言語が含まれていることを示す．一方で出力が 1 の場合，n-gram に異言語が含まれていないことを示す．

3.2.5 *diff* の計算

n-gram に異言語が含まれているかどうかの結果を用いて，検証するトークン列の各トークンに対して，*diff* 値を計算する．*diff* 値は最小値 0，最大値 1 の指標であり，数値大きいほどそのトークンが異言語である可能性が高いことを示す．その結果に基づいて，ユーザーから取得したコードに異言語の慣用句が混入しているかどうかの判定を行う．

diff 値を求める計算式は，以下のようになる．

$$diff = \frac{1}{|N|} \sum_{n \in N} \sum_{gram \in G_n} \frac{gram_{wrong}}{|G_n|} \tag{3.1}$$

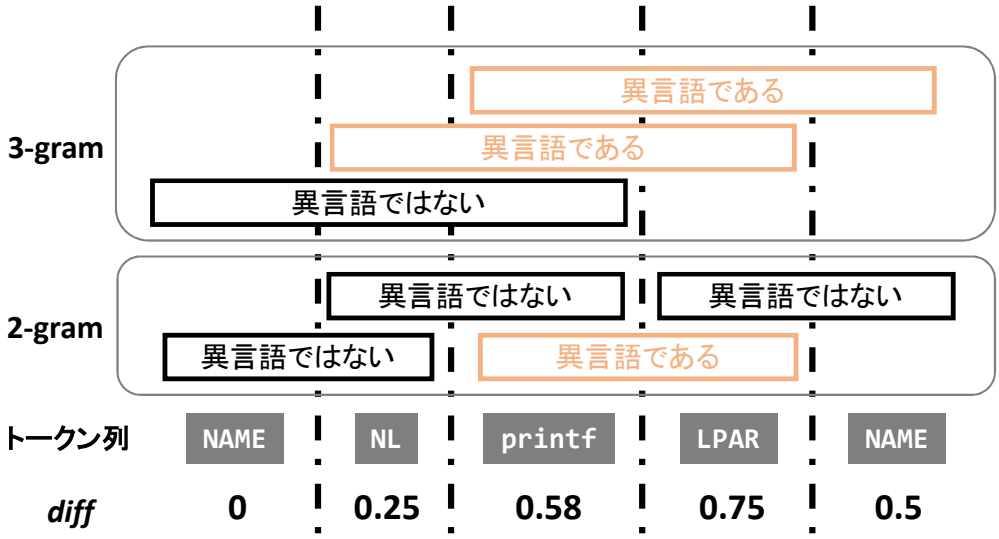


図 3.4. トークン列から生成された 2-gram と 3-gram に対する LSTM の判定の例

ここで，用いる n-gram の長さの集合を N，*diff* 値を求めるトークンが含まれる長さ n の n-gram の集合を G_n ，集合 G_n の元を *gram* ，その *gram* が LSTM によって異言語であると判定されたかを $gram_{wrong}$ と表す．LSTM によって「異言語である」と判定されている場合 $gram_{wrong} = 1$ ，「異言語ではない」と判定されている場合 $gram_{wrong} = 0$ となる．また，集合 A の元の個数は $|A|$ と表す．

例として，図 3.4 のトークン `printf` に対して，*diff* 値を計算する過程を示す．図中のトークン列以外の長方形は n-gram を表し，どの n-gram も LSTM によって「異言語である」もしくは「異言語ではない」と判定されている．まず，用いる n-gram は 2-gram と 3-gram なので，$N = \{2, 3\}$ となる．`printf` が含まれる 2-gram は [`NL printf`] , [`printf LPAR`] の 2 つなので，$G_2 = \{$[`NL printf`],[`printf LPAR`]$\}$，$|G_2| = 2$ となる．同様にして，$G_3 = \{$[`NAME NL printf`],[`NL printf LPAR`],[`printf LPAR NAME`]$\}$，$|G_3| = 3$ となる．図中の 2-gram の判定結果を踏まえると，

$$\sum_{gram \in G_2} \frac{gram_{wrong}}{|G_2|} = \frac{0}{2} + \frac{1}{2} = \frac{1}{2} \tag{3.2}$$

3-gram の判定結果を踏まえると，

$$\sum_{gram \in G_3} \frac{gram_{wrong}}{|G_3|} = \frac{0}{3} + \frac{1}{3} + \frac{1}{3} = \frac{2}{3} \tag{3.3}$$

と計算されるので，トークン printf に対する *diff* 値は以下のように計算される．

$$\begin{aligned} diff &= \frac{1}{|N|} \sum_{n \in N} \sum_{gram \in G_n} \frac{gram_{wrong}}{|G_n|} \\ &= \frac{1}{2}\left(\frac{1}{2} + \frac{2}{3}\right) = \frac{7}{12} \approx 0.58 \end{aligned} \tag{3.4}$$

Listing 3.1. 高確率で Python だと認識されるトークン列の元コード

```
1 a = 0
2 b = 0
3 print(a)
```

diff 値は，検証するトークン列の端のトークンほど LSTM による n-gram の判定の誤判定の影響を受けやすいという欠点がある．これは，トークン列の端のトークンほどそのトークンが含まれている n-gram の数が少なく，1 つの n-gram に対する LSTM の判定結果が，そのトークンの *diff* 値の計算に使われる比重が大きくなるからである．例えば，図 3.4 において，トークン列の右端のトークン `NAME` が含まれる n-gram は 2 つしかない．そのため，1 つの n-gram が 「異言語である」と判定されるだけで，*diff* 値が 0.5 と計算されている．一方で，トークン列の左から 2 番目のトークン `NL` は，4 つ n-gram に含まれる．そのため，1 つの n-gram が 「異言語である」と判定されても，*diff* 値が 0.25 と計算され，右端のトークン `NAME` よりは *diff* 値が小さくなっている．このように，端のトークンは内部のトークンよりも 1 つの n-gram が 判定結果が *diff* 値に与える影響が大きい．そのため，1 つの LSTM の誤判定によって端のトークンの *diff* 値が大きく変わってしまう可能性があり，トークン列全体に対する異言語か否かの判定にも影響が出てしまう恐れがある．

本手法では，前述した問題を解決するために，n-gram 分割の前に偽のトークン列を検証するトークン列の前後に加えて，*diff* 値の計算までの処理を行うことにする．すなわち，n-gram 分割，LSTM による判定，*diff* 値の計算は偽のトークン列を含めて行うことになる．偽のトークン列には，高確率で今書いている言語のものだと認識されるトークン列を用いる．例えば，今書いている言語が Python だった場合には，プログラム 3.1 に示すコードをトークン化した，以下のトークン列を検証するトークン列の前後に加える．

```
NAME EQUAL NUMBER NL NAME EQUAL NUMBER NL print LPAR NAME RPAR
```

この処理によって，*diff* 値の計算をするときに，検証するトークン列のトークンが偽のトークン列を含めた全体のトークン列の内部に位置することになり，LSTM の誤判定の影響を受けにくくなる．次節で述べるコードに異言語の慣用句が混入しているか否かの判定には，検証するトークン列のトークンの *diff* 値だけを用いる．

3.2.6 取得したコード全体に対する判定

取得したコードのすべてのトークンに対して計算された *diff* 値を用いて，コードに異言語の慣用句が混入しているか否かを判定する処理における，最終的な出力を行う．コードに異言語の慣用句が混入していると判定された場合は，コードを LLM に送ってコード翻訳を行う．

まず，*diff* 値を計算したトークンの中で，最も *diff* 値の高いトークンとその *diff* 値 を求める．本論文では，それぞれを $token_{max}$ と $diff_{max}$ と表すことにする．$token_{max}$ は，コードに異言語の慣用句が混入していると判定された場合において，3.3.3 節で説明する修正箇所を

処理で使用される．次に，$diff_{max}$ が閾値 α よりも値が大きいかどうかを確かめる．$diff_{max}$ が閾値 α よりも値が大きい場合は，コードに異言語の慣用句が混入していると判定する．一方で，値が小さい場合は，コードに異言語の慣用句が混入していないと判定する．コードに異言語の慣用句が混入していると判定された場合，コード翻訳と，元のコードと翻訳後のコードから異言語の慣用句部分だけを抜き出す処理を行う．コードに異言語の慣用句が混入していないと判定された場合，コード翻訳の処理に進まない．

3.3　異言語の慣用句を翻訳する手順

本手法では，LLM を用いたコード翻訳と翻訳元コードと翻訳後コードから慣用句部分だけを抜き出す処理を組み合わせることで，異言語の慣用句の翻訳を行う．前節の処理によってユーザーから取得したコードに異言語の慣用句が混入していると判定された場合，LLM を用いてコード翻訳を行う．本論文では，コード翻訳の入力に使われた，ユーザーが書いたコードを「翻訳元コード」，コード翻訳によって出力されたコードを「翻訳後コード」と表す．

異言語の慣用句の翻訳の流れを以下の示す．

1. LLM を用いたコード翻訳
2. レーベンシュタイン距離を計算する動的計画法を用いた，翻訳元コードと翻訳後コードの最適な単語の組み合わせの特定
3. 最適な単語の組み合わせを用いた，翻訳元コードと翻訳後コードの慣用句部分の抽出

本節ではそれぞれの処理の詳細について述べる．

3.3.1　LLM によるコード翻訳

ユーザーから取得したコードを LLM に送りコード翻訳を行うことで，送られてきたコード内に書かれた異言語の慣用句をユーザーが今書いている言語の慣用句に翻訳する．

LLM に送る翻訳元コードは，異言語の慣用句の可能性が高いとされる $token_{max}$ の前後数単語だけでなく，前処理で取得したカーソル周辺の数行のコードをまとめて送る．LLM は，慣用句の周辺文脈を伝えないと，不適切な翻訳を行う可能性があるからである．例えば，Java の `Arrays.sort` メソッドで逆順ソートするコードを間違えて

```
Arrays.sort(src, reverse = true);
```

のように書いてしまったときに，異言語の慣用句の可能性が高い個所として

```
reverse = true
```

が認識されたとする．このコード片のみを LLM に送ってしまった場合，モデルはこのコード片をメソッドの引数としてではなく，単純な代入文だと理解してしまい，

```
boolean reverse = true;
```

翻訳元コード

```
1  a = 0
2  std::cout << a << std::endl
3  a += 1
```

翻訳後コード

```
1  a = 0
2  print(a)
3  a += 1
```

図 3.5. 翻訳元コードと翻訳後コードの例

と翻訳してしまう．このコード片では適切な置換を行うことができない．周辺文脈としてメソッド全体をモデルに送れた場合，モデルはメソッドの引数が間違っていると理解でき，

```
Arrays.sort(src, Collections.reverseOrder());
```

と目的としている翻訳結果が得られる．

LLM にコードとともに送る入力プロンプトに含める指示を以下に示す．

- ユーザーが今書いている言語にコードを翻訳すること
- 自然言語を書かないこと
- 変数などの定義を補完する場合はコードの最初にまとめて書くこと
- 翻訳する部分がない場合は `Correct` とだけ出力すること

変数などの定義を補完する場合はコードの最初にまとめて書く指示は，翻訳元コードと翻訳後コードから慣用句部分だけを抜き出す処理で，LLM のコード補完によって勝手に補完された変数やライブラリなどの定義文を排除するために含める．詳しくは 3.3.3 節で述べる．翻訳する部分がない場合，LLM に `Correct` とだけ出力させることで，入力支援システムの前半部分の異言語の慣用句の混入を判定する処理で誤った判定がおきて，異言語の慣用句の混入がないコードが LLM に渡されても，対処することができる．

3.3.2 最適な単語の組み合わせの特定

翻訳元コードと翻訳後コードの中の最適な単語の組み合わせを求める．この単語の組み合わせは，次の処理において翻訳元コードと翻訳後コードの合致部分，非合致部分を特定するのに用いられる．最適な単語の組み合わせとは，単語の挿入，単語の削除，単語の置換の 3 つの操作を用いて，最小の操作回数で 2 つのコードを一致させるときに，操作を受けない単語の組み合わせである．例えば，翻訳元コードと翻訳後コードが図 3.5 であったとする．これらのコードの最適な単語の組み合わせを表したのが，図 3.6 である．翻訳元コードを最小の操作回数で翻訳後コードに変換するには，図中の矢印で示した操作を行う．青色で示した操作が単語の置

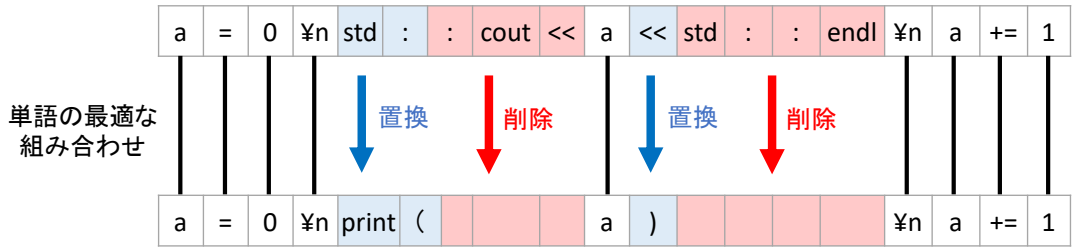


図 3.6. 翻訳元コードと翻訳後コードの最小の操作回数となる操作と最適な単語の組み合わせ

換であり，赤色で示した操作が単語の削除である．これらの操作を受けていない単語，つまり翻訳元コードを翻訳後コードに変換するときに変化していない単語がコード間の最適な単語の組み合わせになる．図 3.6 の中では，黒い線でつながれた単語の集合が，最適な単語の組み合わせになる．

翻訳元コードと翻訳後コードの最適な単語の組み合わせは，以下の手順で求められる．

1. LLM の入力に用いた翻訳元コードと，コード翻訳によって出力された翻訳後コードのそれぞれをトークン単位の単語に分割する．この処理には，3.2.1 節で説明した字句解析器を使用する．
2. レーベンシュタイン距離を求める動的計画法を用いて，翻訳元コードと翻訳後コードを最小の操作回数で一致させるための操作回数を求める．このとき，動的計画法で用いる配列 `DP` のそれぞれの要素の値を更新するために，どの計算をしたかを別の配列 `REC` に記録しておく．2.1.8 節で説明したように，配列 `DP` を更新するときに，`DP[j][i]` の値は，

$$\texttt{DP[j][i]} = \begin{cases} \texttt{DP[j-1][i-1]} & (1) \\ \texttt{DP[j-1][i-1] + 1} & (2) \\ \texttt{DP[j-1][i] + 1} & (3) \\ \texttt{DP[j][i-1] + 1} & (4) \end{cases} \tag{3.5}$$

の (1) から (4) のいずれかで計算される．配列 `REC[j][i]` には，式 3.5 のうち，計算に用いた計算式に対応する番号を記録する．例えば，`DP[j][i]` の値の計算が `DP[j-1][i-1]1 + 1` だった場合，`REC[j][i]` に 2 と記録しておく．`DP[j][i]` の値の計算が `DP[j-1][i]1 + 1` だった場合，`REC[j][i]` に 3 と記録しておく．
3. 配列の最後の要素から，配列 `REC` を用いて計算を遡っていくことで，配列の最後の要素までの計算の道筋を得る．`REC[j][i]` が 1 または 2 だった場合，`REC[j-1][i-1]` に移る．`REC[j][i]` が 3 だった場合，`REC[j-1][i]` に移る．`REC[j][i]` が 4 だった場合，`REC[j][i-1]` に移る．この操作を `REC[j][i]` の `j` または `i` のいずれかの添え字が 0 になるまで繰り返す．

		a	=	0	¥n	std	:	:	cout	<<	a	<<	std	:	:	endl	¥n	a	+=	1
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a	1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
=	2	1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	3	2	1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
¥n	4	3	2	1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
print	5	4	3	2	1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
(	6	5	4	3	2	2	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	7	6	5	4	3	3	3	3	4	5	5	6	7	8	9	10	11	12	13	14
)	8	7	6	5	4	4	4	4	4	5	6	6	7	8	9	10	11	12	13	14
¥n	9	8	7	6	5	5	5	5	5	5	6	7	7	8	9	10	10	11	12	13
a	10	9	8	7	6	6	6	6	6	6	5	6	7	8	9	10	11	10	11	12
+=	11	10	9	8	7	7	7	7	7	7	7	6	7	8	9	10	11	11	10	11
1	12	11	10	9	8	8	8	8	8	8	8	8	7	8	9	10	11	12	11	10

図 3.7. 翻訳元コードと翻訳後コードのレーベンシュタイン距離を求めた時の配列の状態

4. 配列の最後の要素までの計算の道筋の中で，REC[j][i] が 1 である i と j の組 (i,j) を集める．それぞれの (i,j) に対する，翻訳元コードの i 単語目と翻訳後コードの j 単語目が，最適な単語の組み合わせに含まれる単語の組になる．

例として，図 3.5 の翻訳元コードと翻訳後コードから，図 3.6 で示した最適な単語の組み合わせを得るまでの流れを説明する．

まず，翻訳元コードおよび翻訳後コードを字句解析器を用いてトークン単位の単語に分割すると，翻訳元コードは，

```
a, =, 0, \n, std, :, :, cout, <<, a, <<, std, :, :, endl, \n, a, +=, 1
```

翻訳後コードは，

```
a, =, 0, \n, print, (, a, ), \n, a, +=, 1
```

となる．ただし単語間はコンマで区切っている．

次に，これらの単語列に対して，レーベンシュタイン距離を求める動的計画法を行う．これによって，配列 DP の値は図 3.7 の各セルに書かれている値になる．この図の各セルの色は，配列 REC に記録した値に対応している．濃い灰色は 値 1，青色は値 2 ，緑色は値 3，赤色は値 4 を表す．

配列 REC に記録された情報を元に，最後のセルまでの計算を遡っていく．REC[j][i] が 1 または 2 だった場合，REC[j-1][i-1] に移る．すなわち，図 3.7 の濃い灰色のセルと青色のセルからは，左上のセルに移る．同様にして，REC[j][i] が 3 だった場合，REC[j-1][i] に移るので，図中の緑のセルからは上のセルに移る．REC[j][i] が 4 だった場合，REC[j][i-1]

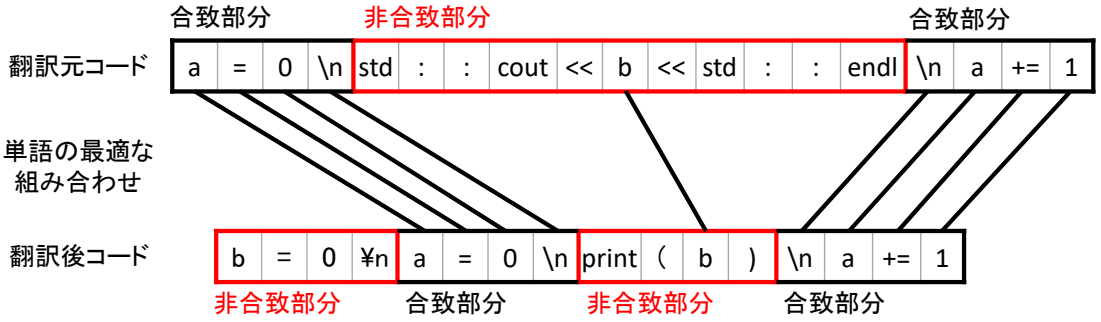


図 3.8. 翻訳元コードと翻訳後コードの差異的な単語の組み合わせと合致・非合致部分

に移るので，図中の桃色のセルからは左のセルに移る．この操作を繰り返すことで，配列の最後の要素までの計算の道筋が得られる．図の中の太い黒線で囲んでいる部分がそれにあたる．

最後に，配列の最後の要素までの計算の道筋の中で，`REC[j][i]` が 1 である要素を見つける．その要素に対応した各コードの単語が，最適な単語の組み合わせに含まれる単語の組である．図 3.7 では，太い黒線で囲まれた要素の中でセルが濃い灰色であるものの，列の一番上に書かれた単語と行の一番左に書かれた単語が，最適な単語の組み合わせに含まれる単語の組となる．これによって，図 3.6 の最適な単語の組み合わせを得られる．

3.3.3　翻訳元コードと翻訳後コードからの慣用句部分の抽出

前節で求めた，最適な単語の組み合わせを用いて，翻訳元コードと翻訳後コードの中で，合致しているコード部分とそうではないコード部分を特定する．そのあと，それぞれのコード部分の対応関係と，3.2.6 節で求めた $token_{max}$ の位置から，翻訳元コードの慣用句部分と翻訳後コードの慣用句部分，すなわち異言語の慣用句と今書いている言語に翻訳された慣用句をそれぞれ抽出する．入力支援としては，コードエディタ上でこの異言語の慣用句にエラー破線を表示し，修正コードとして翻訳された慣用句を提示することになる．

最適な単語の組み合わせの組が連続で 3 組以上続くコード部分を合致する部分とする．それ以外のコード部分を非合致部分とする．この処理の例を図 3.8 に示す．図の中の上のコードが翻訳元コード，下のコードが翻訳後コードを表し，二つのコードの単語同士を結ぶ黒線が最適な単語の組み合わせとなる．この例だと，各コードの

```
a = 0 \n
```

と

```
\n a += 1
```

のコード部分が，最適な単語の組み合わせの組が連続で 3 組以上続くコード部分であるので，合致部分となる．それ以外の部分は非合致部分となる．翻訳元コードの 10 単語目と翻訳後コードの 11 単語目の `b` は最適な単語の組み合わせの組であるが，前後の単語が最適な単語の

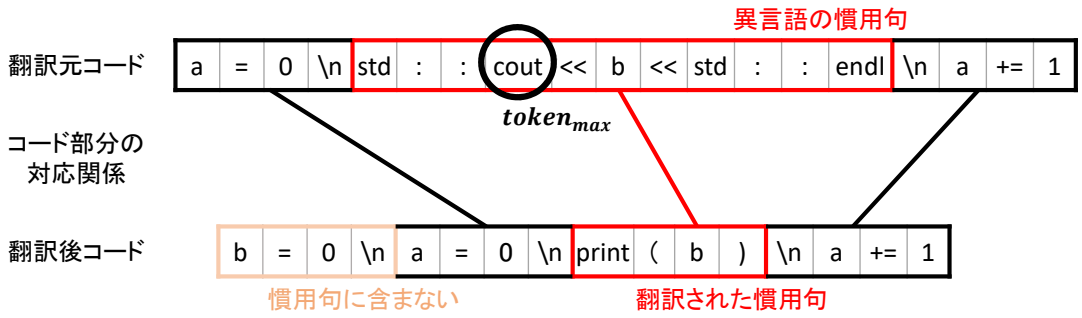


図 3.9. 異言語の慣用句と翻訳された慣用句の抽出

組み合わせの組ではないので，非合致部分に含まれる．

これらの合致部分，非合致部分の対応関係と $token_{max}$ の位置を用いて，翻訳元コードの慣用句部分と翻訳後コードの慣用句部分をそれぞれ特定する．まず，最適な単語の組み合わせから，翻訳元コードの合致部分と翻訳後コードの合致部分の対応関係を求める．次に，翻訳元コードと翻訳後コードの非合致部分のうち，2 組の対応関係にある合致部分の間に挟まれた非合致部分の組をすべて探し，これらは対応関係にあると定める．翻訳元コードの非合致部分のうち，$token_{max}$ に対応する単語を含むものを異言語の慣用句，その非合致部分と対応関係にある翻訳後コードの非合致部分が，翻訳された慣用句となる．先ほどの例で合致部分と非合致部分を特定した翻訳元コードと翻訳後コードから，異言語の慣用句と翻訳された慣用句を抽出する処理を 図 3.9 に示す．図の中で，コード部分を結ぶ黒線が合致部分の対応関係を示し，赤線が非合致部分の対応関係を示す．図で示すように，翻訳元コードの `cout` が $token_{max}$ に対応する単語である場合，その単語を含む非合致部分

```
std :: cout << b << std :: endl
```

が異言語の慣用句となり，それと対応関係にある翻訳後コードの非合致部分

```
print(b)
```

が翻訳された慣用句となる．

この手法によって，LLM のコード翻訳によって勝手に補完された変数やライブラリの定義文などを排除することができる．例えば，図 3.9 の例でも，LLM が補完した変数定義文である

```
b = 0 \n
```

を翻訳された慣用句から排除できている．これは，合致部分の対応関係と非合致部分の対応関係を求めることで，翻訳元コードの特定のコード部分が，翻訳後コードのどの部分にあたるのかを認識できるようになったからである．

ただし，LLM による補完が，翻訳された慣用句の周辺に存在する場合，排除できない可能性がある．例えば，図 3.9 の翻訳後コードが

```
a = 0 \n b = 0 \n print(b) \n a += 1
```

であった場合，

```
b = 0 \n
```

と

```
print(b) \n
```

の間に合致部分がないので，2 つのコード片は同じ非合致部分に含まれてしまう．すなわち，LLM による補完が，翻訳された慣用句部分に含まれてしまう．

この問題を防ぐために，3.3.1 節で説明したとおり，LLM がコード翻訳で変数などの定義を補完する場合は，コードの最初にまとめるようにプロンプトに指示文を含める．この指示によって，翻訳元コードの異言語の慣用句にあたる部分の前に，合致部分となるコード部分が存在すれば，LLM よって勝手に補完された変数やライブラリの定義文をまとめて排除することができる．

第 4 章

実験

本章では，前章の提案手法に基づいたシステムの実装とその性能を評価するために行った実験について述べる．

4.1 LSTM の事前学習

3.2.4 節で述べた解析手法の重要部分である LSTM の学習を行い，その精度を検証する．この LSTM は入力に最大 20 トークンのトークン列を取り，出力としてトークン列が対象の言語で書かれたものか否かを判定する．我々は，Python，Java，C++ の 3 つの言語のコードから変換されたトークン列がどの言語のものであるか判定するタスクを教師あり学習で LSTM に学習させた．各言語ごとに別々の LSTM を用意し，入力されたトークン列がそれぞれの言語のものであるか否かの 2 値分類で判定させた．本論文では，LSTM が判定の対象としている言語をターゲット言語と呼ぶことにする．例えば，トークン列 が Python で書かれたコードから変換されたものであるか否かを判定する LSTM のターゲット言語は Python となる．本節では，LSTM の学習の詳細と学習後のモデルの精度について述べる．

4.1.1 Skip-gram の事前学習

LSTM の Embedding レイヤーで用いる各トークンに対するベクトルを計算するために，Skip-gram モデルを学習させた．学習によって，各トークンに対して 200 次元のベクトルを取得した．データセットには，Python ，Java ，C++ の各言語についてそれぞれ，GitHub [*1] から 10,000 ファイルのソースコード，競技プログラミングのデータセットである Project CodeNet [*2] から 2,000 ファイルのソースコードを集めた．このデータセットは，LSTM の事前学習に使用したデータセットとは別で集めたデータセットである．

ソースコードのトークン列への変換は，3.2.1 節で説明した Python の字句解析器を元にした字句解析手法を用いた．また，3.2.2 節で述べた通り，字句解析器で標準で使用されるトーク

[*1] https://github.com

[*2] https://developer.ibm.com/exchanges/data/all/project-codenet/

表 4.1. データセットとして使用する n-gram の最低登場回数

n-gram	最低登場回数	n-gram	最低登場回数
1-gram	18	11-gram	8
2-gram	18	12-gram	8
3-gram	18	13-gram	8
4-gram	18	14-gram	8
5-gram	18	15-gram	8
6-gram	13	16-gram	3
7-gram	13	17-gram	3
8-gram	13	18-gram	3
9-gram	13	19-gram	3
10-gram	13	20-gram	3

ンとは別に，予約語，標準ライブラリの関数名やメソッド名，その他のよく使用する識別子を個別のトークンとして定義した．予約語と標準ライブラリの関数名やメソッド名は， Python [*3]，Java [*4]，C++ [*5] それぞれの公式ドキュメントなどから取得する．その他のよく使用する識別子は，各言語の 12,000 ファイルのソースコードに対して識別子の出現回数をそれぞれ調べ，出現回数がその言語に対して上位 100 個ほどに含まれる識別子を個別のトークンとする．ただし，`i` や `j` などの 1 文字の識別子は個別のトークンにしない．

4.1.2　データセット

Python ，Java ，C++ の各言語についてそれぞれ，GitHub から 10,000 ファイルのソースコード，Project CodeNet から 2,000 ファイルのソースコードを集めた．それらのソースコードを 3.2.1 節の字句解析手法でトークン化したあと，ファイル全体のトークン列を 1-gram から 20-gram で分割した．その後，ほとんど使われない n-gram 表現を削除するため，各 n-gram に対して，表 4.1 で示す最低登場回数未満の n-gram はデータセットから排除した．これによって，Python の n-gram の集合，Java の n-gram の集合，C++ の n-gram の集合が得られた

ターゲット言語で書かれているとみなす n-gram には正解ラベル 1 を，ターゲット言語とは異なる言語で書かれているとみなす n-gram には正解ラベル 0 を与える．そのために，以下の条件で正解ラベルを各 n-gram に与えた．

- ターゲット言語の n-gram の集合に含まれる n-gram それぞれに正解ラベル 1 を与

[*3] `https://docs.python.org/ja/3/reference/`
[*4] `https://www.oracle.com/jp/java/technologies/documentation.html`
[*5] `https://en.cppreference.com/w/`

表 4.2. 各 LSTM の学習と検証で使用したデータセットの数

ターゲット言語	学習用のデータセット数	テスト用のデータセット数
Python	5256072	1314018
Java	4518236	1129560
C++	9474958	2368740

える．

- ターゲット言語以外の言語の n-gram の集合に含まれる n-gram のうち，ターゲット言語の n-gram の集合に含まれない n-gram には正解ラベル 0 を与える．

例えば，以下のコードを

```
for (int i=0;
```

トークン化したトークン列は，

```
for LPAR int NAME EQUAL SEMI
```

Java の n-gram の集合と C++ の n-gram の集合には含まれるが，Python の n-gram の集合には含まれないと考えられる．この場合，Java と C++ をターゲット言語にする LSTM ではこのトークン列に正解ラベル 1 が与えられ，学習に用いられる．一方で，Python をターゲット言語にする LSTM では正解ラベル 0 が与えられ，学習に用いられる．

作成したデータセットは，正解ラベル 1 を持つ n-gram のデータ数と正解ラベル 0 を持つ n-gram のデータ数は同数になるようにデータセットを調整した．それぞれのデータ数のうち，少ない方のデータ数に合わせるように，多い方のデータを無作為に削除した．これは LSTM の判定が偏らないようにするためである．

以上の手順で作成したデータセットを学習用のデータセットとテスト用のデータセットに無作為に分割した．データセットの 8 割を学習用に，2 割をテスト用に使用した．Python，Java，C++ のそれぞれをターゲット言語とする LSTM の学習とテストに用いた n-gram の数を表 4.2 に示す．

4.1.3 学習の詳細とモデルの精度

Python の ライブラリ `PyTorch` を用いて，LSTM の学習と検証を行った．学習の詳細を以下に示す．

損失関数

損失関数には，`Pytorch` で使用できる `BCEWithLogitsLoss` [*6] を用いた．この損失関数は，シグモイド関数とバイナリ交差エントロピー (Binary Cross-Entropy) を 1 つに

[*6] https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html

組み合わせたものである．2 値分類予測での `BCEWithLogitsLoss` による損失 L の計算式は以下のようになる．

$$l_n = -w_n[y_n log(\sigma(\hat{y}_n)) + (1 - y_n) log(1 - \sigma(\hat{y}_n))] \tag{4.1}$$

$$L = \sum_{n=1}^{N} l_n \tag{4.2}$$

ただし，N 個のデータセットのうち，n 番目のデータに対する予測値を $\hat{y}_n$，正解ラベルを y_n ，計算で用いる重みを w_n で表す．また，σ はシグモイド関数を表す．

バッチサイズ

バッチサイズを 1024 として，ミニバッチ学習を行った．ミニバッチ学習とは，学習で用いるデータセットをいくつかのグループにデータを無作為に小分けし，グループごとに学習を行う手法である．それぞれのグループごとに損失の計算と勾配の更新を行う．

エポック数

最大エポック数を 50 回と設定し，各エポックで学習とテストデータを用いた検証を交互に行った．過学習を防ぐために，2 エポック連続でテストデータの正答率が減少した場合，学習を終了した．

図 4.1 は，Python，Java，C++ をそれぞれターゲット言語にしている 3 つの LSTM の，各エポックでの学習データと検証データに対する正答率の変化を表している．また，図 4.2 は，各エポックでの学習データと検証データに対する損失の変化を表している．図中の青色の実線が学習データ，赤色の破線がテストデータにそれぞれ対応している．どの LSTM も最終エポックでは，テストデータに対する正答率が 98 % 以上を示し，高い精度を示した．

4.2　入力支援システムの検証

入力支援システムを実装し，パラメータの最適化と判定精度の検証，修正精度の検証を行った．この検証は，今書いている言語が Python ，混入する可能性のある慣用句は Java もしくは C++ で書かれたものと仮定して行った．そのため，異言語の慣用句の混入を判定する処理で用いる LSTM は Python がターゲット言語であるものを使用した．また，LLM のコード翻訳の翻訳先の言語は Python に設定した．LLM の API には GPT-3.5-turbo [*7] を用いた．

まず，異言語の慣用句の混入を判定する処理における判定精度に，n-gram の長さと閾値 α の値がどのように関係するかを調べ，判定に使用する n-gram の長さと閾値 α を最適化した．次に，入力支援システム全体での異言語の慣用句の混入に対する判定精度を検証した．最後に，入力支援システムが実際に行った修正を，我々の期待する修正と比べることで評価した．

4.2.1　評価指標

本節では，実験に用いた評価指標について述べる．

[*7] https://platform.openai.com/docs/models/gpt-3-5

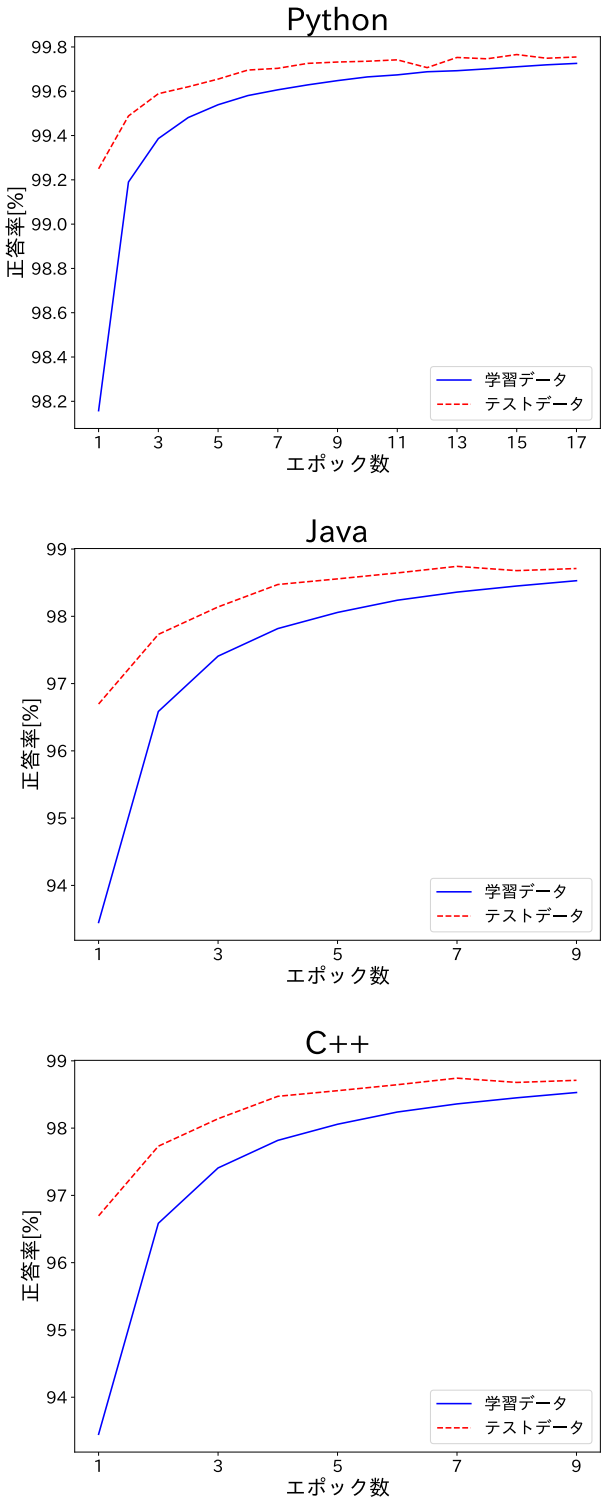


図 4.1. 各エポックでの学習データとテストデータそれぞれに対する LSTM の正答率．上のグラフが Python，真ん中のグラフが Java，下のグラフが C++ をそれぞれターゲット言語にする LSTM の結果を示す

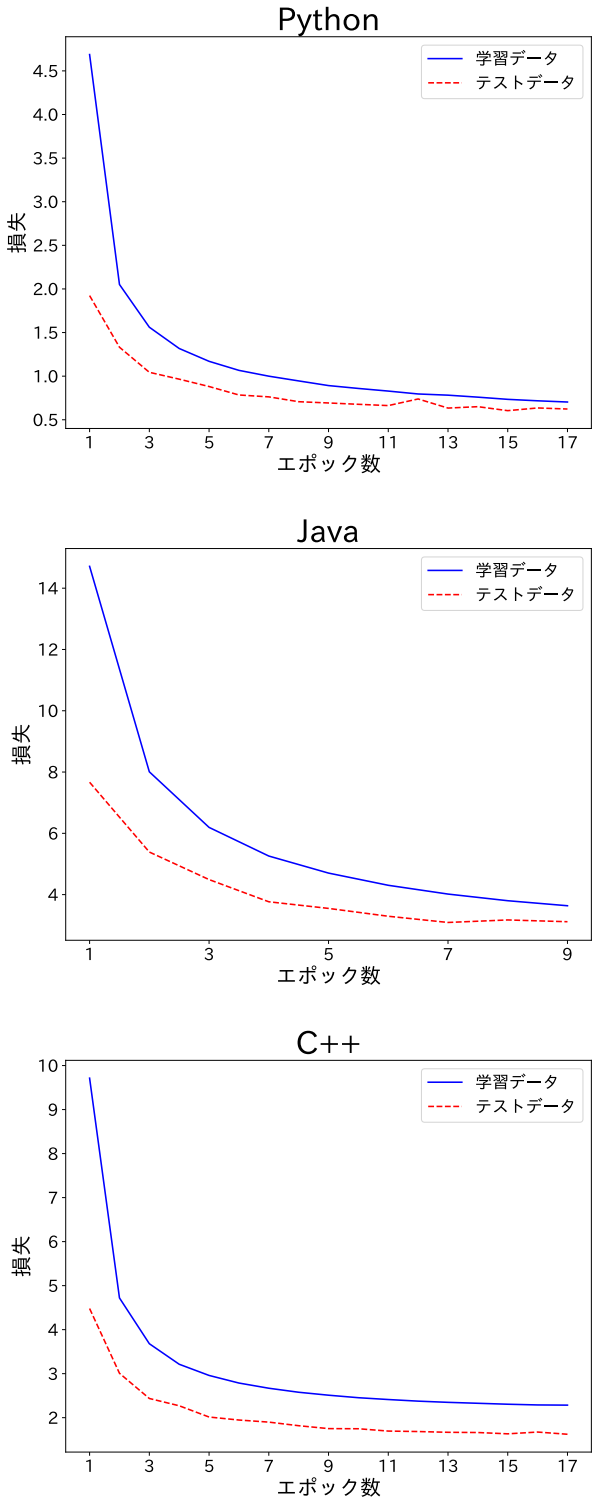


図 4.2. 各エポックでの学習データとテストデータそれぞれに対する LSTM の損失．上のグラフが Python，真ん中のグラフが Java，下のグラフが C++ をそれぞれターゲット言語にする LSTM の結果を示す

判定のための評価指標

機械学習の 2 値の分類問題の分類結果および実際の状態は，「陽性」と「陰性」にわけることができる．本研究の，異言語の慣用句が混入しているか否かの判定も，2 値の分類問題であるため，本論文では以降，「異言語の慣用句が混入している」と判定することを陽性，「異言語の慣用句が混入していない」と判定することを陰性と表現することにする．

2 値の分類問題における判定は，判定結果と実際の状態によって以下の 4 つに区分される．

True Positive (TP)　陽性であると判定して，実際に陽性である場合
True Negative (NP)　陰性であると判定して，実際に陰性である場合
False Positive (FP)　陽性であると判定して，実際は陰性である場合
False Negative (FN)　陰性であると判定して，実際は陽性である場合

これらを用いた評価指標には，以下のようなものがある．

Accuracy

単純な評価指標として，すべての判定のうち，正解した割合を示す．[0, 1] の範囲で値をとり，値が大きいほど良い．以下の計算で求められる．

$$Accuracy = \frac{|TP| + |TN|}{|TP| + |TN| + |FP| + |FN|} \tag{4.3}$$

Precision

陽性と判断されたもののうち実際に陽性だったものの割合を示す．[0, 1] の範囲で値をとり，値が大きいほど False Positive が抑えられていることを示す．以下の計算で求められる．

$$Precision = \frac{|TP|}{|TP| + |FP|} \tag{4.4}$$

Recall

実際に陽性だったもののうち陽性と判定できたものの割合を示す．[0, 1] の範囲で値をとり，値が大きいほど False Negative が抑えられていることを示す．以下の計算で求められる．

$$Recall = \frac{|TP|}{|TP| + |FN|} \tag{4.5}$$

Specificity

実際に陰性だったもののうち陰性と判定できたものの割合を示す．[0, 1] の範囲で値をとり，値が大きいほど False Positive が抑えられていることを示す．以下の計算で求められる．

$$Specificity = \frac{|TN|}{|TN| + |FP|} \tag{4.6}$$

F1 score

$Precision$ と $Recall$ の調和平均を示す．[0, 1] の範囲で値をとり，値が大きいほど良

い．以下の計算で求められる．

$$F1_{score} = \frac{2Precision \cdot Recall}{Precision + Recall} \tag{4.7}$$

このうち，本研究では主に Recall と Specificity を使用する．

BLEU Score

BLEU Score は「モデルの出力する翻訳文が，人間の翻訳者が翻訳した参照訳に近いほど，モデルの翻訳精度が高い」という前提のもとで測る指標である．[0, 1] の範囲で値をとり，値が大きいほど翻訳精度が高いことを示す．BLEU Score の計算式を以下に示す．

$$BLEU = BP_{BLEU} exp(\sum_{n=1}^{N} w_n log(p_n)) \tag{4.8}$$

$$p_n = \frac{\text{翻訳文中の } n-gram \text{ のうち，参照訳中にある } n-gram \text{ 数}}{\text{翻訳文中の全 } n-gram \text{ 数}} \tag{4.9}$$

$$w_n = \frac{1}{n} \tag{4.10}$$

BP_{BLEU} は翻訳文が参照訳よりも長かった場合のペナルティを表す．w_n は n-gram に対する重みを表す．

4.2.2　データセット

本章の検証を行うために，以下の 3 つのデータセットを作成した．

陰性データセット

無作為な Python のソースコードから 5 から 6 行ほどのコードを抜き出したデータセット

陽性データセット

Python のソースコード中の 1 つの慣用句だけが異言語の慣用句，つまり Java もしくは C++ の同じ処理をする慣用句に置き換わっている 5 から 6 行ほどのコードのデータセット

修正データセット

陽性データセットの各コードの異言語の慣用句が，Python の慣用句に修正されているデータセット

異言語慣用句の混入を判定する処理の最適化，および入力支援システムの判定精度の検証には陰性データセットと陽性データセットを使用した．入力支援の修正精度の検証には陽性データセットと修正データセットを使用した．

陰性データセットは，Github と Project CodeNet から無作為に選んだ 5000 ファイルの Python ソースコードそれぞれから，連続した 5 から 6 行ほどのコードを無作為に抽出することで作成した．

Listing 4.1. 複雑な慣用句を含むコード例

```
words = [["Java", "Python"],
         ["Stream", "FlatMap"],
         ["Example", "Code"]]

# 2 次元リストを1 次元リストに変換
flat_list = words.stream().flatMap(List::stream)
                          .collect(Collectors.toList());

print("Flat List:", flat_list)
```

陽性データセットと修正データセットに対応するような公開されているデータセットは我々の知る限り存在しない．そのため，これらのデータセットを自作した．まず，陽性データの中で使用する異言語の慣用句を定義した．本研究では，慣用句の長さと複雑さの違う以下の 3 種類の慣用句を集めた．

簡単な関数やメソッド

print や append など，言語によって関数名やメソッド名が異なるだけ，もしくは少し文法を変えるだけで良い，簡単に書きかえることができる慣用句．

制御構造

for 文 や if 文，サブルーチンの定義などの制御構造．

複雑な慣用句

Java の Stream API など，翻訳のときに関数名やメソッド名を書きかえるだけではなく，文法を全体的に変える必要がある慣用句．例えば，プログラム 4.1 のコードの 6 行目と 7 行目などが，複雑な慣用句である．

これらの慣用句が書かれた Python のソースコードを GitHub もしくは Project CodeNet から探して, 慣用句が書かれた行とその前後数行を抜き出した．もし, 慣用句が書かれた Python のソースコードが見つからなければ，その慣用句が含まれる 5 から 6 行ほどのコード片を作成した．これらのコード片の集合を修正データセットとして扱う．また，修正データセットのそれぞれのコードの中で， 慣用句にあたる部分を Java もしくは C++ の慣用句に書き換えた．これを陽性データセットとして扱う．定義した 89 の慣用句に対して陽性データセットと修正データセットを作成した．各慣用句に対して，最大 20 ファイルを作成した．合計 859 のコード片を持つ 陽性データセット と 修正データセット を作成した．

4.2.3 異言語慣用句の混入を判定する処理の最適化

入力支援システムの前半部分にあたる，異言語慣用句の混入を判定する処理における判定の精度と，使用する n-gram，閾値 α の関係性をそれぞれ検証した．データセットには陽性デー

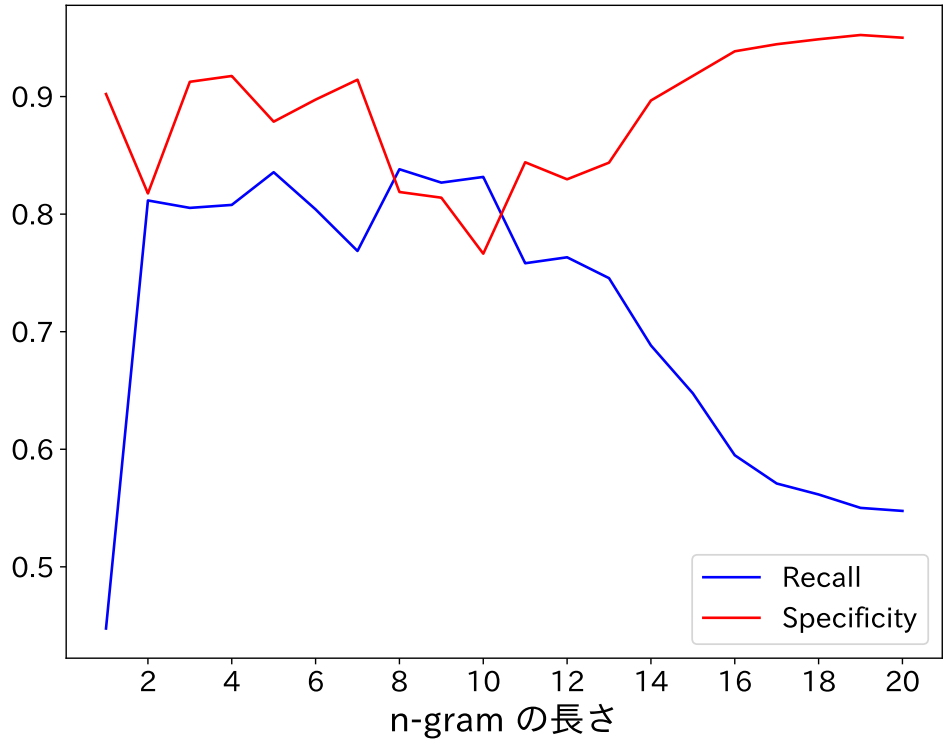


図 4.3. n-gram 分割に使用した n-gram の長さと，判定結果の Recall と Specificity の関係性

タセットと陰性データセットを合わせたものを使用した．

本検証では，異言語慣用句の混入を判定する処理を用いて，Java や C++ の慣用句を含んだ陽性データセットからのコード片を「陽性」，Python でしか書かれてない陰性データセットからのコード片を「陰性」として，入力コードの陽性・陰性を判定させた．

使用する n-gram の長さの最適化

異言語慣用句の混入を判定する処理において使用する最適な n-gram の長さを調べた．異言語慣用句の混入を判定する処理の中で，3.2.3 節で説明した n-gram 分割のときに，特定の長さの n-gram だけを用いた．例えば，2-gram だけに分割する，もしくは 5-gram だけに分割する，というように分割した．どの判定処理においても，閾値 α は 0.2 に設定した．処理の中で使用した n-gram の長さによってどの程度モデルの精度が変わるかを検証した．

使用した n-gram 長さに対する，判定結果によって計算された Recall と Secificity を図 4.3 に示す．Recall は，実際に陽性だったもののうち陽性と判定できたものの割合を示す．Secificity は，実際に陰性だったもののうち陰性と判定できたものの割合を示す．Secificity は多少の変化はあったが，n-gram の長さに関係なく 0.75 以上の高い数値を示した．一方で，Recall は n-gram の長さが 2 から 12 の間は 0.75 以上の高い数値を示したが，それ以外の場

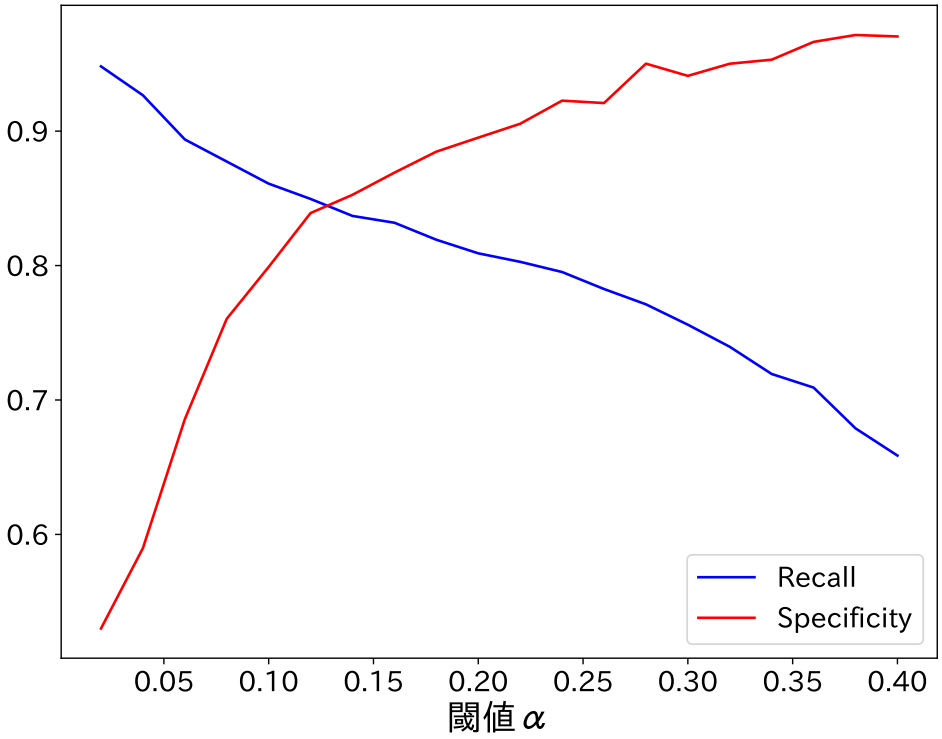


図 4.4. 閾値 α と，判定結果の Recall と Specificity の 関係性

合では著しく数値を下げた．

この結果に基づき，以降の検証では， n-gram の分割の処理において，最小 2-gram から最大 12-gram に分割することにする．今回の入力システムでは，異言語の慣用句が含まれていないのに異言語の慣用句が含まれていると判定してしまう False Positive の場合では，次の LLM の処理で異言語の慣用句が含まれていなかったと判定し直すことができる．一方で，異言語の慣用句が含まれているのに異言語の慣用句が含まれていないと判定してしまう False Negative の場合では，もう一度判定し直す機会が存在しない．そのため，False Negative をどれだけ抑えられているかを示す Recall を，高くすることが求められる．

閾値 α の最適化

異言語慣用句の混入を判定する処理において，判定のときに用いる閾値 α の最適な値を調べた．閾値 α は異言語が混入しているかどうかを判定するための数値的な基準である．閾値 α が小さければ異言語の可能性を表す *diff* 値に対して判定が厳しくなり，大きければ *diff* 値に対して判定が緩くなる．すなわち，閾値 α が小さければ False Positive が多くて，Specificity が低くなる．閾値 α が大きければ False Negative が多くて，Recall が低くなる．そのため，Recall と Specificity のどちらもが低くない閾値 α を調べた．

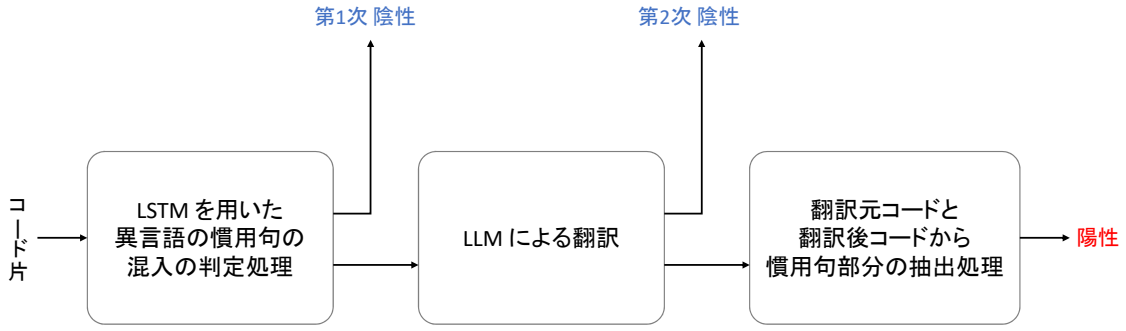


図 4.5. 入力支援システム全体でのコードの陽性・陰性の判定を行う仕組みの概要

閾値 α を 変化させながら，異言語慣用句の混入を判定する処理を行い，それぞれの閾値 α に対しての， Recall と Secificity を計算した．図 4.4 にその結果を示す．前述したようにこの判定においては Recall の方を重要視する．α が 0.10 のとき，Recall は 0.86，Specificity は 0.80 とどちらも高い数値で判別できる．そのため，以降の検証では，$\alpha = 0.10$ に設定する．

4.2.4　入力支援システムの判定精度の検証

入力支援システム全体で，異言語の慣用句の混入の判定をどの精度でできているかの検証を行った．図 4.5 に入力支援システム全体での，コードの陽性・陰性の判定を行う仕組みの概要を示す．入力支援システムでは，前半部分の LSTM を用いた処理が，異言語の慣用句の混入を判定する主な処理である．しかし，その判定で陽性だと判定されて LLM に送られた後にも，送られてきたコードが陽性であるかどうかの判定を行う．入力支援システム全体が行った判定において，LSTM を用いた異言語の慣用句の混入の判定処理において陽性と判定され，さらに LLM による判定でも陽性と判定されたら，システム全体の判定で陽性となる．すなわち，LSTM を用いた異言語の慣用句の混入の判定処理において陽性と判定されても，LLM による判定では陰性と判定された場合，システム全体の判定で陰性になる．LSTM を用いた異言語の慣用句の混入の判定処理で陰性と判定されたものを「第 1 次陰性」，LLM による判定で陰性と判定されたものを「第 2 次陰性」と表すことにする．

本節では，まずすべてのデータセットに対する判定の評価を行った結果を示す．データセットには陽性データセットと陰性デートセットを用いる．次に，陽性データセットを含まれる慣用句に応じて，4.2.2 で説明した，「簡単な関数やメソッド」，「制御構造」，「複雑な慣用句」の 3 つの慣用句の種類に分けて，それぞれのグループに対する判定の評価を行った結果を示す．評価指標としては Recall と Secificity とともに，LSTM を用いた異言語の慣用句の混入の判定に対する Recall として Recall(LSTM)，Secificity として Secificity(LSTM)，LLM による判定だけに対する Recall として Recall(LLM)，Secificity として Secificity(LLM) を用い

表 4.3. すべてのデータセットに対する判定の評価

評価指標	値
Recall	0.75
Recall(LSTM)	0.86
Recall(LLM)	0.87
Secificity	0.94
Secificity(LSTM)	0.80
Secificity(LLM)	0.67

た．それぞれの計算式は以下のようになる．

$$Recall(LSTM) = \frac{|TP| + |FN_2|}{|TP| + |FN|} \tag{4.11}$$

$$Secificity(LLM) = \frac{|TN_1|}{|TN| + |FP|} \tag{4.12}$$

$$Recall(LLM) = \frac{|TP|}{|TP| + |FN_2|} \tag{4.13}$$

$$Secificity(LLM) = \frac{|TN_2|}{|TN_2| + |FP|} \tag{4.14}$$

ただし，FN_1 と FN_2 は False Negative のうち，第 1 次陰性だったものと第 2 陰性だったものを表す．TN_1 と TN_2 は True Negative のうち，第 1 次陰性だったものと第 2 陰性だったものを表す．

まず，すべてのデータセットに対する判定の評価を表 4.3 節に示す．Recall は 0.75 となり，前節で検証した LSTM を用いた異言語の慣用句の混入の判定処理だけに対する Recall より低くなった．一方で，Secificity は 0.94 となり，LSTM を用いた異言語の慣用句の混入の判定処理だけに対する Secificity より低くなった．これは LLM による判定では，LSTM を用いた異言語の慣用句の混入の判定で True Positive だったものを False Negative に変える，もしくは False Positive だったものを True Negative に変えることしかないためである．このことは，Recall と Recall(LSTM)，Recall(LLM) の関係と，Secificity と Secificity(LSTM)，Secificity(LLM) の関係を式にした以下の式からも読み取れる．

$$Recall = Recall(LSTM)Recall(LLM) \tag{4.15}$$

$$Secificity = Secificity(LSTM) + (1 - Secificity(LSTM))Secificity(LLM) \tag{4.16}$$

また，Secificity(LLM) は 0.67 と比較的低い数値を示した．このことから，LSTM を用いた異言語の慣用句の混入の判定が False Positive，すなわち異言語の慣用句が混入していないのに，混入していると判定されたコードは，LLM でも同じ判定をされやすいことがわかった．入力支援システム全体の判定で False Positive だったコード片の例をプログラム 4.2 に示す．

Listing 4.2. 入力支援システム全体の判定で False Positive だったコード例

```
            "Monitoring %s (%s) for changes in %s: %s",
            remote,
            get_remote_url(remote),
            ref,
```

表 4.4. 異言語の慣用句の種類ごとに対する判定の評価

慣用句の種類	評価指標	値
簡単な関数やメソッド	Recall	0.77
制御構造	Recall	0.91
複雑な慣用句	Recall	0.55
簡単な関数やメソッド	Recall(LSTM)	0.89
制御構造	Recall(LSTM)	0.99
複雑な慣用句	Recall(LSTM)	0.68
簡単な関数やメソッド	Recall(LLM)	0.86
制御構造	Recall(LLM)	0.92
複雑な慣用句	Recall(LLM)	0.81

Listing 4.3. LSTM を用いた異言語の慣用句の混入の判定で，簡単な関数やメソッドの異言語の慣用句に対して False Negative だったコード例

```
n=int(input())
nums=[]
for i in range(n):
    nums.add(int(input()))
count=0
for i in range(n):
```

このように，引数の一部分，もしくはリスト定義の一部分である各行がカンマで区切られたコード片が，False Positive に多く含まれていた．

次に陽性データセットの各コード片を，含んでいる異言語の慣用句の種類を「簡単な関数やメソッド」,「制御構造」,「複雑な慣用句」の 3 つにわけることによって，3 つのグループに分けた．それぞれのグループに対する判定の評価を表 4.4 に示す．LLM による判定は，その多くを LLM の性能に依存する．そのため，ここでは主に，LSTM を用いた異言語の慣用句の混入の判定に対する False Negative の考察を行う．

簡単な関数やメソッド

簡単な関数やメソッドは，入力支援システム全体の Recall として 0.77 とまずまずの結

果を示した．Recall(LSTM) は 0.89 であり，LSTM を用いた異言語の慣用句の混入の判定では，異言語の慣用句の混入に対して，9 割程度検出することを示した．LSTM を用いた異言語の慣用句の混入の判定で False Negative となったコード片の例をプログラム 4.3 に示す．このコード片の 4 行目にリスト `nums` に要素を追加する処理として，Java などでリスト型に要素を追加する `add` メソッドが書かれているが，Python ではリスト型に要素を追加するのに `append` メソッドを用いるので，この `nums.add` は異言語の慣用句である．このコード片が False Negative になった要因として，Python では `set` 型に要素を追加するメソッドとして，`add` メソッドが存在することがあげられる．LSTM はトークン化されたオブジェクトの元々の型を考慮せずに，前後数トークンの情報だけで判定を行う．そのため，`add` メソッドは異言語の慣用句ではないとして，False Negative の判定が出てしまう．

制御構造

制御構造は，入力支援システム全体の Recall として 0.91 と非常に高い結果を示した．Recall(LSTM) も 0.99 であり，LSTM を用いた異言語の慣用句の混入の判定ではほぼすべての異言語の慣用句を検出できた．

複雑な慣用句

複雑な慣用句は，入力支援システム全体の Recall として 0.55 と 3 種類の慣用句の中で，最も低い結果を示した．Recall(LSTM) も 0.68 であり，LSTM を用いた異言語の慣用句の混入の判定では，異言語の慣用句の混入に対して，7 割程度を検出することを示した．LSTM を用いた異言語の慣用句の混入の判定で False Negative となったものの中には，C++ で用いられる `pop_back` メソッドが用いられたコードがいくつか存在した．これは，`pop_back` を用いた C++ のソースコードが Skipgram モデルのデータセットの中に含まれていなかったため，`pop_back` が 個別のトークンではなく，識別子トークンに変換されてしまっているためだと考えられる．

4.2.5 入力支援の修正の精度の検証

入力支援システムの判定で，True Positive であったコード片に対して行った異言語の慣用句のコード翻訳の性能を検証した．評価指標には BLEU Score を用いた．データセットには，陽性データのうち，入力支援システムの判定で True Positive であったコード片に，3.3 節で説明した異言語の慣用句の翻訳手法を用いて修正したものを BLEU Score の「翻訳文」として用いる．陽性データセットのそのコード片に対応する修正データセットのコード片を BLEU Score の「参照訳」として用いる．

図 4.6 はすべての True Positive のコード片に対して BLEU Score を計算したときの値の分布を表す．翻訳文が参考訳と一致する場合は BLEU Score は 1 となる．このような翻訳文の参照ペアは 30 % 存在した．また，翻訳文の参照ペアのうち 74 % が，BLEU Score が 0.7 以上であった．このように，異言語の慣用句のコード翻訳の性能が高いことが示された．

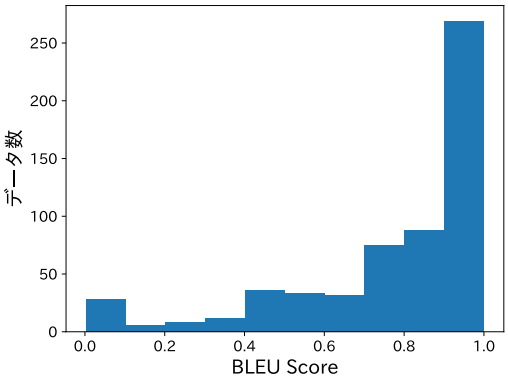
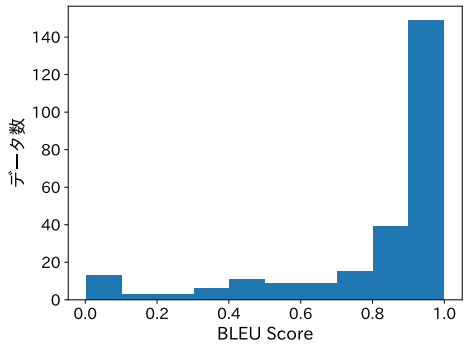
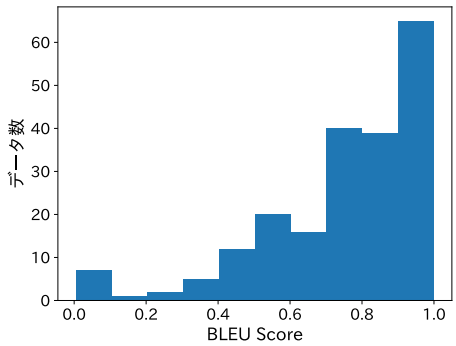


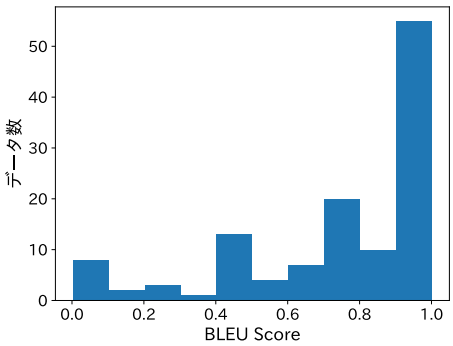
図 4.6. True Positive の各コード片を入力支援で修正したときのコードと，それぞれに対応する修正データセットのコードの間の BLEU スコア



a. 簡単な関数やメソッド



b. 制御構造



c. 複雑な慣用句

図 4.7. 各種類の慣用句の True Positive のコードを入力支援で修正したときのコードと，それぞれに対応する修正データセットのコードの間の BLEU スコア．a は慣用句の種類が簡単な関数やメソッドのときのグラフ，b は制御構造のときのグラフ，c は複雑な慣用句のときのグラフを表す．

また，図 4.7 は各種類の慣用句の True Positive のコード片に対して BLEU Score を計算したときの値の分布を示す．BLEU Score が 0.7 以上であるものが，簡単な関数やメソッドだと 79 %，制御構造だと 70 %，複雑な慣用句だと 69 % とどの慣用句の種類に対しても比較的精度の高い翻訳を行うことができた．翻訳文が参考訳と一致する場合，すなわち BLEU Score が 1 になるものは， 単な関数やメソッドだと 40 %，制御構造だと 18 %，複雑な慣用句だと 31 % となった．

4.3 実験のまとめ

本節では，4.1 節，および 4.2 節で行った実験結果をまとめる．

まず, 異言語の慣用句の混入を判定する処理で用いる LSTM の事前学習を行った．Python, Java，C++ の n-gram を用いて，入力される n-gram がターゲット言語で書かれたものであるかどうかを判定する教師あり学習を行った．結果として，ターゲット言語がどの言語でも，LSTM は 98 % 以上の精度で判定を行えることが確認できた．

次に，異言語の慣用句の混入を判定する処理で分割する n-gram の長さと，判定に用いる閾値 α の最適化を行った．特定の n-gram の長さを用いた場合における異言語の慣用句の混入への判定精度を調べ，2 から 12 の n-gram を用いることにした．同様に，閾値 α に対する異言語の慣用句の混入への判定精度を調べ，閾値 α は 0.10 に設定することにした．

入力支援システム全体の異言語の慣用句の混入を判定する性能を検証した．今書いている言語は Python であるとして判定した．全体の精度としては Recall が 0.75，Secificity が 0.94 を示した．すなわち，異言語の慣用句が混入に対して， 75 % ほどを認識することができた．また，異言語の慣用句が混入していない入力に対しては， 94 % ほどの判定性能を示した．そのため，改善する部分はあるが，全体的に悪くない精度で判定が行えることを示した．また，異言語の慣用句の種類別に入力支援システム全体の判定性能を調べた．簡単な関数やメソッドに対しては Recall が 0.77，制御構造に対しては 0.91，複雑な慣用句に対しては 0.55 を示した．

異言語の慣用句が混入していない入力に対して 6 % ほど誤判定が起きた要因の一つは，引数の一部分，もしくはリスト定義の一部分である各行がカンマで区切られたコード片のいくつかを，異言語の慣用句が混入していると判定してしまったためである．また，異言語の簡単な関数やメソッドや複雑な慣用句を認識する割合が低かった要因としては，異言語の関数やメソッドの名前が Python で用途の違う関数やメソッドと同じ名前であった場合や，Skip-gram モデルの学習データセットに含まれていなかったために個別のトークンとして扱われていなかったことが挙げられる．今後の研究では，これらの要因への対処が求められる．

最後に，先ほどの入力支援システム全体の異言語の慣用句の混入の判定性能において，True Positive となったデータに対して，コード翻訳と慣用句抽出を行い，元のコードの異言語の慣用句部分を翻訳した慣用句に置換した修正コード (翻訳文) が，我々が理想とする修正コード (参照文) にどれだけ近いかを BLEU Score を用いて検証した．結果として，翻訳文と参照文のうち，完全一致したものが 30 % 存在した．また，74 % 以上が BLEU Score 0.7 以上と高

い数値を示した．慣用句の種類ごとにも BLEU Score を確かめたが，BLEU Score が 0.7 以上の割合が最も少なかった複雑な慣用句でも 69 % であり，どの慣用句に対しても高精度な翻訳を行うことができることを示した．この検証では，判定処理において True Positive だったものしか検証できておらず，False Negative となったデータに対しての修正精度を調べられていない．今後，それらのデータの検証を行う必要がある．

第 5 章

まとめと今後の課題

5.1 まとめ

本論文では，異言語の慣用句を入力してしまった場合に，その修正候補をユーザーに提示する入力支援システムを開発するために，LLM を用いたコード翻訳に，異言語の慣用句の混入を判定する処理と，翻訳元コードと翻訳後コードから慣用句部分を抽出する処理を組み合わせる手法を提案した．この入力支援の実現により，複数言語を使い分けるような開発環境で，言語間の慣用句の違いによってユーザーが起こす混乱に対して，ユーザーを支援することができる．

本手法を用いて実装した入力支援システムに対して，コードが異言語の慣用句を含むか否かを判定するタスクの性能を実験で検証した．結果として，制御構造などの慣用句は非常に高い精度で判定できたが，関数やメソッドの判定や，Java の Stream API などの複雑な慣用句の判定では一定数誤判定があった．これらは主に，言語間で役割が違っても同じ名前である関数・メソッドや，Skip-gram モデルの学習のデータセットに含まれていなかったために個別のトークンとして定義されなかった関数・メソッドによるものだと考えられる．

また，入力支援システムが提示する修正候補が，我々が理想としている修正候補にどの程度近いかを検証した．検証結果から，どのような慣用句に対しても，比較的高精度な修正を行えることが分かった．

5.2 今後の課題

本研究で行った実験では，Python，Java，C++ の 3 つの言語を対象に LSTM の事前学習を行い，今書いている言語を Python，異言語で混入する言語が Java と C++ と仮定して入力支援システムの検証を行った．そのため，他の言語を書いている場合や他の言語が混入する場合の性能はわからない．情報の少ないマイナーな言語や関数型言語などを用いた場合，性能が下がる可能性も考えられる．今回の実験では，提案手法を用いて開発した入力支援の性能を簡易的に検証するために，一般的に広く使われている少数の言語だけを対象にしたが，将来的に上記に示すような言語を含めて，様々な言語に対して対応できる手法なのかを検討する必要

がある．

本研究で実装した入力支援システムによる，コードが異言語の慣用句を含むことを検知するタスクは，異言語の制御構造への検知は良い精度を示したが，異言語の関数やメソッドを認識できない例が複数あった．その中で，Skip-gram モデルの学習データセットに含まれていないために，個別のトークンとして扱うべき関数名やメソッド名に対応するベクトルが生成されず，ただの識別子トークンのベクトルがそれらの関数名やメソッド名に対応するベクトルになったものがある．本研究の実験では，Skip-gram の学習および LSTM の学習には各言語につき，12000 ファイルのソースコードをデータセットとして利用したが，様々なタスクを行うソースコードをより多く集めてデータセットとして用いることで，より汎用性のあるモデルが開発できると考えられる．

本研究では，異言語の慣用句が混入している陽性データセットと，その慣用句を今書いている言語に修正した修正データセットは，既存のデータセットがないために，手作業で作成した．そのため，我々が定義した 89 の慣用句に対してのみ実験を行っており，それ以外の慣用句に対しての性能評価を行えていない．入力支援システムの性能を更に厳密に検証するには，より様々な異言語の慣用句が混入した，大規模なデータセットを作成する必要がある．

発表文献と研究活動

(1) 宮原 和也. 山崎 徹郎. 千葉 滋. 慣用句単位のコード翻訳を応用したプログラミング支援の提案. 日本ソフトウェア科学会 第 40 回大会. 2023.09.12-13.

参考文献

[1] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, Vol. 9, No. 8, pp. 1735–1780, 1997.

[2] S. Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München. 1991.

[3] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, Vol. 529, No. 7587, pp. 484–489, January 2016.

[4] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. 2013.

[5] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. 2013.

[6] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, Vol. abs/1409.3215, , 2014.

[7] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, Vol. abs/1412.3555, , 2014.

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, Vol. abs/1706.03762, , 2017.

[9] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. 2020.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. 2019.

[11] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael

Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. 2023.

[12] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. 2020.

[13] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. Lamda: Language models for dialog applications. 2022.

[14] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models. 2022.

[15] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. 2022.

[16] OpenAI, :, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. Technical report, 2023.

[17] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. 2023.

[18] VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, Vol. 10, p. 707, 1966.

[19] Github Copilot. https://github.com/features/copilot.

[20] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman t, et al. Evaluating large language models trained on code. 2021.

[21] Sara Verdi. Inside GitHub: Working with the LLMs behind GitHub Copilot. https://github.blog/2023-05-17-inside-github-working-with-the-llms-behind-github-copilot/.

[22] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[23] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. An empirical study on the usage of BERT models for code completion. *CoRR*, Vol. abs/2103.07115, , 2021.

[24] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample.

Unsupervised translation of programming languages. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20. Curran Associates Inc., 2020.

[25] Guillaume Lample, Myle Ott, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. Phrase-based & neural unsupervised machine translation, 2018.

謝辞

本研究を進めるにあたって，研究方針や発表に対して親身になって相談に乗ってくださり，毎週のミーティングでご指導ご鞭撻いただきました千葉 滋教授には深く感謝いたします．また，そのミーティングに毎回出席してくださり，研究に必要なアイディアを助言していただいた山崎 徹郎氏にも感謝いたします．