東京大学
情報理工学系研究科 創造情報学専攻
修士論文

教師ラベルなし単言語学習データのみでの cross-language コードクローン検出の試み

Attempts on detecting cross-language code clones using only monolingual training data without labels

劉 宇澤
Yuze Liu

指導教員　千葉 滋 教授

2021 年 1 月

概要

近年, 同じプロジェクトでも複数の言語に関わることが増え, 教師あり学習で cross-language コードクローンを検出する手法が研究され始めている．しかしながら，教師あり学習はデータセットの質に依存するため，良いデータセットが存在しない今の状況では，実応用にはまだ現実的ではない．そこで，本研究は教師データが必要ない教師なし学習での cross-language コードクローンの検出を試みた．まだ研究されたことのないタスクであるため，その可能性を追求するに，二つの仮説を立てた．そして，二つの仮説を元に，LSTM を使った autoencoder や Transformer を使った言語モデルとそれぞれの改善法について実験した．学習データとして GitHub からアルゴリズムに関係する Java と Python のコードを集め，評価は LeetCode の解答を使って行った．結果はかなり高い精度で cross-language のコードクローンを検出することができた．また，教師あり学習手法と比較した．

Abstract

In recent years, the same project has been increasingly involved in multiple languages, and the detection of cross-language code clones is beginning to be studied. However, since supervised learning depends on the quality of the dataset, it is not yet possible to apply it to practical use in the current situation where good datasets does not exist. In this thesis, we present unsupervised learning of cross-language code clone detection that does not require labeled training data. Since this is the first attempt in this field, we made two hypothesis to support it will work. Then, based on this two hypotheses, we experimented with autoencoder using LSTM and language model using Transformer and the improvement approaches for each. Java and Python codes related to the algorithm were collected from GitHub as training data, and evaluation was performed using the solutions of LeetCode. As a result, we were able to detect cross-language code clones with fairly high performance. We also compared it with the supervised learning method.

目次

第 1 章

はじめに

コードクローンとは，ソースコードの中に存在する互いに一致，もしくは類似したコード片である．コードクローンはソフトウェアの開発および保守に悪影響を与える恐れがある．その理由の一つは，あるコードを修正したなら，そのコードのコードクローンを全てに対して修正を行う必要があるからである．コードクローンが存在する原因の一つはコピーアンドペーストである．新しい機能を追加するときに，すでにその機能と類似した機能を実装した場合は，その部分からコピーアンドペーストをすることが多い．もし，コピー元にバグが含まれていた場合，そのバグをたくさん増やすことになる．特に大規模ソフトウェアはコードクローンの数が多い可能性が高いため、保守作業においてコードクローンが大きな問題となりやすい。同一言語でのクローン検出はたくさん研究されている [1, 2, 3, 4, 5, 6].

近年，同じプロジェクトでも複数の言語に関わることが増えている．例えば，プロトタイプコードとプロダクトコードは違う言語を使うことがある．プロトタイプコードは Python などのスクリプト言語を使い，実際のプロダクトコードは C++ や Java などの速い言語で実装する．また，アプリケーションが動く環境によって，使える言語が限られている．同じ機能のソフトウェアでも，iOS アプリケーションは Swift や Objective C を使い，Android アプリケーションは Java や Kotlin で実装し，ブラウザアプリケーションは JavaScript で書く．こういう背景の中に cross-language のコードクローンが存在する可能性が増えている．cross-language のコードクローンでもソフトウェアの開発および保守に悪影響を与える恐れがあるから，検出することが大切だ.

Cross-language コードクローンの検出は研究され始めている [7, 8, 9, 10, 11, 12]．そして，昨年に，教師データを使って，教師あり学習で cross-language コードクローンを検出する論文が二つ出た [11, 12] が，あまり性能が良くない．論文は二つとも教師データとして，AtCoder[*1]や Google CodeJam[*2]などの競技プログラミングのデータを使って，学習した．競技プログラミングのコードはコーディングスタイルや解決方法に差が大きすぎるため，良いデータではない．教師あり学習はデータセットの質に依存するため，良いデータセットが存在

[*1] https://atcoder.jp

[*2] https://www.go-hero.net/jam/10/languages/0

しない今の状況では，まだ性能が高くならない．そして，教師あり学習は学習されていない他の領域のパターンに弱い．競技プログラミングで学習されたモデルは，競技プログラミングのコードに使えるが，実用的の領域には使えない．実用的の領域にはまだ学習データが存在していないため，実応用にはまだ現実的ではない．

そこで，本研究は教師データが必要ない教師なし学習での cross-language コードクローンの検出を試みた．教師なし学習は教師あり学習に比べて，主に以下三つの利点が期待できる：

1)　**性能に期待できる．**教師あり学習は教師データの質に依存するため，まだ良い教師データが存在しない今の状況では，性能は高くない．一方，教師ラベルなしの単言語データはたくさん存在する．教師なし学習の精度が期待できます．
2)　**汎用性に期待できる．**実用的の領域にはまだ教師ラベル付きの学習データが存在していないため，教師あり学習は実用に応用することはできない．一方，教師ラベルなしの単言語データはどの領域でもたくさん存在している．実用にすぐ応用できる．
3)　**モデルに通す計算量が低い，大規模の検出にも応用できる．**教師あり学習はモデルに通す計算量は $\mathrm{O}(n^2)$ に対して．教師なし学習はモデルに通す計算量を $\mathrm{O}(n)$ に抑えることができる．

教師なし学習による cross-language コードクローン検出は我々が知る限り，まだ研究されたことのないタスクであるため，その可能性を追求するに，二つの仮説を立てた．

仮説 1　自然言語と違い，プログラミング言語は違う言語でもたくさんの共通点があり，これらの共通点をもとに，言語の間の違う部分もベクトル空間上の近い位置に学習できる．
仮説 2　プログラミング言語は分布仮説に従っていて，英語ベースで付けられた変数名や関数名は隠しラベルの役目を果たす．

二つの仮説を元に，合計五つの教師なし学習手法を提案した．

手法 1　LSTM を使った autoencoder．二つの言語を一つのモデルに学習する．
手法 2　手法 1 の上に，言語ごとに違う decoder を使う．
手法 3　手法 1 の上に，言語ごとに違うスタートトークンを使う．
手法 4　Transformer を使った言語モデル，二つの言語を一つのモデルに学習する．
手法 5　手法 4 の上に，コードの分散表現に言語に関わる特徴を見つけて減らす．

学習データとして GitHub[*3]からアルゴリズムに関係する Java と Python のコードを集め，評価は LeetCode[*4]の解答を使って行った．また，教師あり学習手法と比較した．結果，全ての手法は教師あり学習手法より大幅に上回ることができた．

本論文の残り部分の構成を紹介する．2 章では cross-language コードクローンの研究背景や機械学習に関する前提知識について述べる．それから，cross-language コードクローンに関

[*3] https://github.com
[*4] https://leetcode.com

わる関連研究や教師あり学習による検出の問題点について述べる．3 章では，提案した教師なし学習手法の利点と教師なし学習を可能にする二つの仮説を紹介する．4 章では，集めた学習データと評価データを紹介し，未知語を解決するための手法 BPE を紹介する．それから，提案した五つの手法と実験結果を紹介する．その後，教師あり学習手法と比較し，サンプルを挙げて議論する．最後に，5 章で本研究のまとめと今後の課題について述べる．

第 2 章

背景

2.1 cross-language コードクローンについて

2.1.1 コードクローン

コードクローンとは，ソースコードの中に存在する互いに一致，もしくは類似したコード片である．コードクローンは類似度によって，4 つのタイプに分類することができる．タイプ１は最も類似度が高く，タイプ４は一番類似度が低いコードクローンである．これから例を合わせて，四つのタイプを紹介する．

タイプ１

タイプ１のコードクローンは，空白やタブ，コメントなどの字句解析する時に無視できる要素を除いて，完全に一致するコードクローンである．以下は Two Sum を例として説明する．Two Sum とは，整数のリストとターゲット数が与えられ、整数のリストの中から足し合わせるとターゲット数になる 2 つの数を見つけ、そのインデックスを返せという問題である．解答の例は：

```
public int[] twoSum(int[] nums, int target) {
    for (int i = 0; i < nums.length; i++) {
        for (int j = i + 1; j < nums.length; j++) {
            if (nums[j] == target - nums[i]) {
                return new int[] { i, j };
            }
        }
    }
}
```

そして，以下のコードは上の例のタイプ１コードクローンである：

```
public int[] twoSum(int[] nums, int target) {
```

```
    //comment
    for (int i = 0; i < nums.length; i++) {
    for (int j = i + 1; j < nums.length; j++) {
    if (nums[j] == target - nums[i]) {return new int[] { i, j };}}}}
```

タイプ１クローンは字句解析をすれば同じトークン列になるので，トークン列を比較すれば簡単に判別できる．

タイプ２

タイプ２のコードクローンとは，変数名や関数名などのユーザ定義名，また変数の型などの一部の予約語のみが異なるコードクローンである．以下のコードは例のタイプ２コードクローンである：

```
public int[] twoSum(int[] nums, int target) {
    for (int a = 0; a < nums.length; a++) {
        for (int b = a + 1; b < nums.length; b++) {
            if (nums[b] == target - nums[a]) {
                return new int[] { a, b };
            }
        }
    }
}
```

タイプ３

タイプ３はタイプ２のコードクローンの違いに加えて，文の挿入や削除，変更が行われたコードクローンである．例えば，例のタイプ３コードクローンは：

```
public int[] twoSum(int[] nums, int target) {
    for (int i = 0; i < nums.length; i++) {
        for (int j = i + 1; j < nums.length; j++) {
            a = nums[i]
            b = nums[j]
            if (a + b == target) {
                return new int[] { i, j };
            }
        }
    }
}
```

タイプ 4

タイプ 4 のコードクローンは，同じ機能だが，違う方法やアルゴリズムで実装されているコードである．例の Two Sum の問題は，ハッシュテーブルを使っても解決できる．ハッシュテーブルの解答と例のブルートフォースの解答はタイプ 4 コードクローンである：

```
public int[] twoSum(int[] nums, int target) {
    Map<Integer, Integer> map = new HashMap<>();
    for (int i = 0; i < nums.length; i++) {
        int complement = target - nums[i];
        if (map.containsKey(complement)) {
            return new int[] { map.get(complement), i };
        }
        map.put(nums[i], i);
    }
}
```

コードクローンはソフトウェアの開発および保守に悪影響を与える恐れがある．その理由は，あるコードを修正したなら，そのコードのコードクローンを全てに対して修正を行う必要があるからである．コードクローンが存在する原因の一つはコピーアンドペーストである．新しい機能を追加するときに，すでにその機能と類似した機能を実装した場合は，その部分からコピーアンドペーストをすることが多い．コピーアンドペーストの後に，細かい修正を加えれば，新しい機能が完成できる．コピーアンドペーストは新しい機能を素早く実装できる一方，保守作業を困難にした．もし，コピー元にバグが含まれていた場合，そのバグをたくさん増やすことになる．特に大規模ソフトウェアはコードクローンの数が多い可能性が高いため、保守作業においてコードクローンが大きな問題となりやすい。

同一言語でのクローン検出はたくさん研究されている．既存の検出技術はコードクローンをどのように検出するのかによって，大まかに以下の 5 つに分類することができる．

- 行単位の検出
- トークン単位の検出
- 抽象構文木を用いた検出
- プログラム依存グラフを用いた検出
- メトリクスやフィンガープリントなど，その他の技術を用いた検出

2.1.2　cross-language コードクローン

cross-language コードクローンとは違う言語の間に存在するコードクローンである．例として，Two Sum の問題を Java で実装したコードは：

```
public int[] twoSum(int[] nums, int target) {
    for (int i = 0; i < nums.length; i++) {
        for (int j = i + 1; j < nums.length; j++) {
            if (nums[j] == target - nums[i]) {
                return new int[] { i, j };
            }
        }
    }
}
```

そして，同じ問題を Python で実装したコードは：

```
def twoSum(self, nums: List[int], target: int) -> List[int]:
    for i, num in enumerate(nums):
        for j in range(i+1, len(nums)):
            if target - num == nums[j]:
                return [i, j]
```

この２つのコードは cross-language コードクローンである．

近年，同じプロジェクトでも複数の言語に関わることが増えている．例えば，プロトタイプコードとプロダクトコードは違う言語を使うことがある．プロトタイプコードは Python などのスクリプト言語を使い，実際のプロダクトコードは C++ や Java などの速い言語で実装する．また，アプリケーションが動く環境によって，使える言語が限られている．同じ機能のソフトウェアでも，iOS アプリケーションは Swift や Objective C を使い，Android アプリケーションは Java や Kotlin で実装し，ブラウザアプリケーションは JavaScript で書く．

こういう背景の中に cross-language のコードクローンが存在する可能性が増えている．cross-language のコードクローンでもソフトウェアの開発に悪影響を与える恐れがあるから，検出することが大切だ．

例として，同じ機能のソフトウェアが複数言語で実装される場合は，一つ言語のコードが修正される時，他の言語の同じ部分も修正される必要がある．多言語のプロジェクトは通常複数のチームで協力して開発ことが多い．開発者は自分が専門している言語の部分だけ開発している．一つ言語のコードが修正された時，開発者は自分が詳しくない別の言語のコードからコードクローンを探し出す必要がある．システムアーキテクチャや別の言語の予備知識を知る必要があるため，同一言語のコードクローンよりも時間消費やコストが大きい．

また，ウェブアプリケーションはマイクロサービスアーキテクチャを採用することがある [11]．サービスごとに違うチームで違い言語を使って開発するため，サービスの間に機能的に重複するコードが存在する可能性がある．これらのコードは単一責任原則を違反しているから，cross-language コードクローンを検出する技術を使って，避けるべきだ．

2.2　機械学習の前提知識

2.2.1　ニューラルネットワーク

単純なニューラルネットワークは図 2.1 で示した.

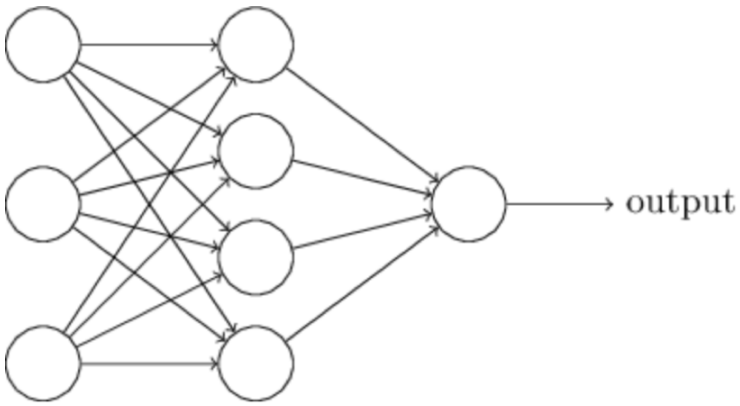

図 2.1. 単純なニューラルネットワーク

一番左の層は入力層と呼ばれ，一番右の層は出力層と言います，そして，真ん中の層は隠れ層である．各層はニューロンと呼ばれたユニットで構成される．１つのニューロンは図 2.2 で示した.

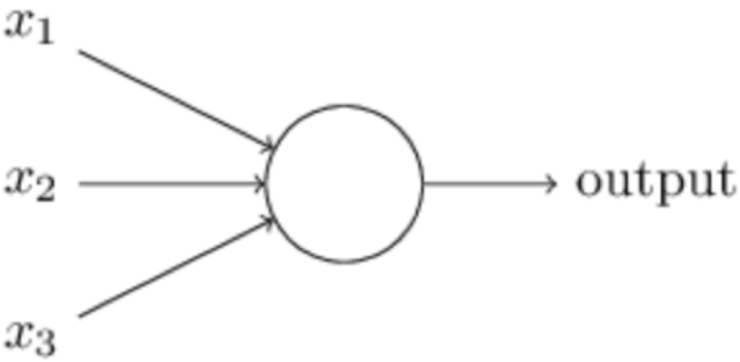

図 2.2. ニューロン

ニューロンは $x_1, x_2, \ldots$ といった入力を取る．そして，それぞれの入力に対して，重み $w_1, w_2, \ldots$ を持ち，またニューロン全体に対するバイアス b と呼ばれる値を持っている．出力は $z = \sum_j w_j x_j + b$ になる．さらに出力に活性化関数をかける．単純な活性化関数はシグモイド関数などがあります．シグモイド関数は式 2.1で定義される.

$$\sigma(z) \equiv \frac{1}{1+e^{-z}} \tag{2.1}$$

最後の出力は式 2.2 になる．

$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)} \tag{2.2}$$

ニューラルネットワークは学習することによって，重みやバイアスなどのパラメータが最適化されます．最適のパラメータとはコスト関数を最小化としたパラメータである．コスト関数の一例として，MSE(mean squared error) がある．MSE を使ったコスト関数は式 2.3 で示した．

$$C(w,b) \equiv \frac{1}{2n}\sum_x \|y(x) - output\|^2 \tag{2.3}$$

ここで w はネットワーク中の全ての重み，b は全バイアス，n は訓練入力の総数，$output$ は入力が x の時にネットワークから出力されるベクトル，$y(x)$ は期待されるネットワークからの出力である．

コスト関数の最小化に，勾配降下法が使われている．勾配降下法の更新規則は式 2.4 と式 2.5 で示す．

$$w_k \to w'_k = w_k - \eta\frac{\partial C}{\partial w_k} \tag{2.4}$$

$$b_l \to b'_l = b_l - \eta\frac{\partial C}{\partial b_l} \tag{2.5}$$

式の中に C はコスト関数である．ηは学習率と呼ばれるハイパーパラメータである．

2.2.2 RNN と LSTM

RNN(Recurrent Neural Network) は過去の状態を保持することができるニューラルネットワークである．そのため，文章のような時間上に連続的な情報を扱うことができる．RNN は過去の状態を保持するために，ループする経路が持つ．データがループすることによって，情報は絶えずに更新されることになる．RNN レイヤを図 2.3 で示す．

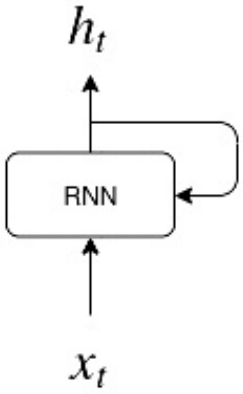

図 2.3. RNN レイヤ

このループ構造を時間上に展開することで，右方向に伸びるニューラルネットワークへ変形させることができる．図 2.4 は RNN の展開図を示す．

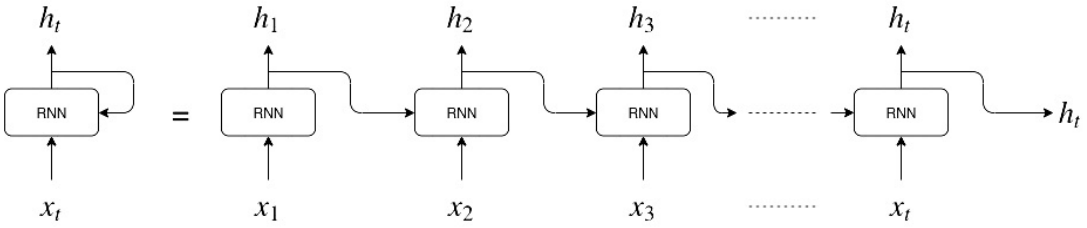

図 2.4. RNN レイヤの展開図

各時刻の RNN レイヤはそのレイヤへの入力と 1 つ前の RNN レイヤからの出力を受け取っている．この 2 つの情報を元に，その時刻の出力が計算される．計算式は式 2.6 で示す．

$$h_t = tanh(h_{t-1}W_h + x_tW_x + b) \tag{2.6}$$

式の中に h_t は時刻 t の時の隠れ状態である．RNN の隠れ状態はこの式で時間ごとに更新される．RNN レイヤの計算グラるを図 2.5 で示す．

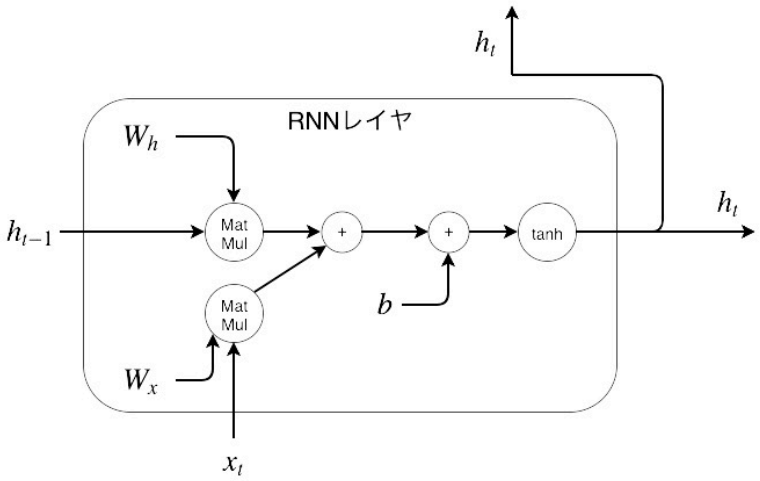

図 2.5. RNN レイヤの計算グラフ

RNN は通常のニューラルネットワークと同様に誤差逆伝播法を用いることができる．時間方向の誤差逆伝播法は BPTT(Backpropagation Through Time) と呼ばれる．BPTT の問題点として，各時刻の中間データをメモリに保持することが必要であるため，長い時系列データを学習する際に消費するコンピュータリソースが膨大になってしまう．また，時間サイズが長くなると，逆伝播時の勾配が不安定になる．この問題を解決するため，Truncated BPTT が使われる．Truncated BPTT は逆伝播のつながりを適当な長さで切り取り，切り取られたネットワーク単位で学習を行う．

Truncated BPTT を使われた RNN は逆伝播の長さが制限されでしまう．また，長い文章だと，勾配が途中で弱まってしまい，ほとんど何も情報を持たなくなり，重みパラメータは更新されなくなることがある．時間を遡るに連れて勾配が小さいくなく勾配消失もしくは大きくなる勾配爆発のどちらかに辿ってしまう．勾配爆発は勾配クリッピングと呼ばれる手法を用いることで解決できる．勾配の L2 ノルムが閾値を超えてしまった場合，勾配が修正される．勾配消失を解決するために，新しいアーキテクチャの LSTM(Long short-term memory)[13] が使われる．

LSTM と RNN のインターフェースの比較は図 2.6 で示す．

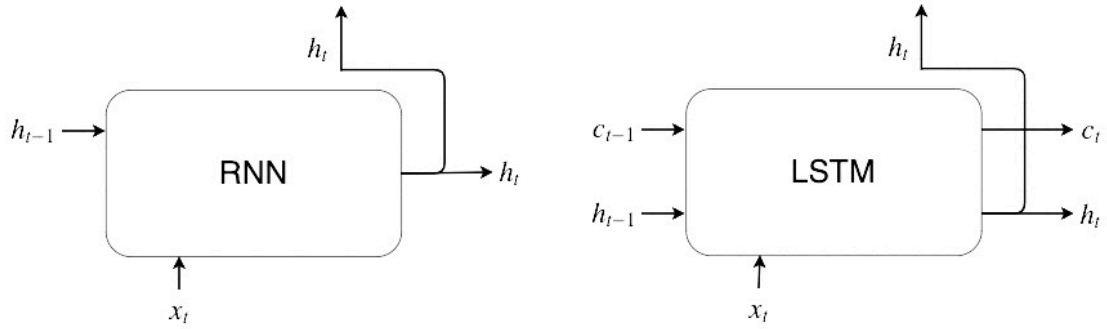

図 2.6. RNN と LSTM の比較

大きな違いとして，LSTM レイヤには c という経路が存在している．c は記憶セルと呼ばれ，LSTM の記憶をコントロールしている．

理解しやすいため，LSTM の計算グラフを図 2.7 で示す．

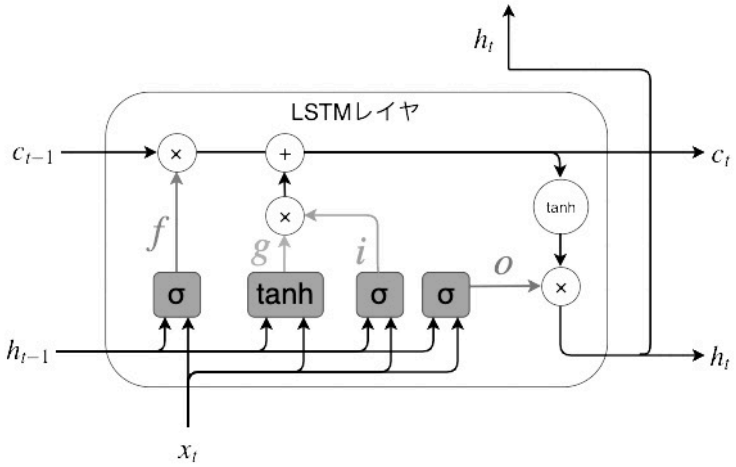

図 2.7. LSTM の計算グラフ

LSTM は 3 つのゲート（図の中に f, i, o で示した）と新しい情報を覚えされるための新しい記憶セル（図の中に g で示した）があります．

- forget ゲート（f）:記憶セルに対してなにを忘れるかを示す．
- 新しい記憶セル（g）：新しく覚えるべき情報を記憶セルに追加する．
- input ゲート（i）：新しく追加する情報の取捨選択を行う．
- output ゲート（o）：次時刻の隠れ状態を調整する．

学習するときに，記憶セルの勾配は逆方向で過去へ伝えていく．Forget ゲートは忘れべきと判断した記憶セルの要素に対して，その勾配の要素を小さくする．そして，忘れてはいけないと判断した要素に対して，その勾配をそのまま過去へ伝わる．そのため，記憶セルの勾配は，覚えておくべき情報に対して勾配消失を起こさずに伝播することができる．

2.2.3 Transformer

Transformer[14] は，計算速度向上のため，RNN を使わずに，Attention メカニズムを使って並列計算を可能にするモデルである．

RNN は時刻ごとのアウトプットは次の時刻のインプットであるため，時刻の順番通りに計算しなければならない．従って，並列計算ができず，GPU などを使っても，その能力をフルに活用できない．そこで，RNN を使わない Attention メカニズムのみを使った Transformer が提案された．Transformer により，より少ない学習時間で，精度の高いモデルができた．

Transformer の構造は図 2.8 で示す．

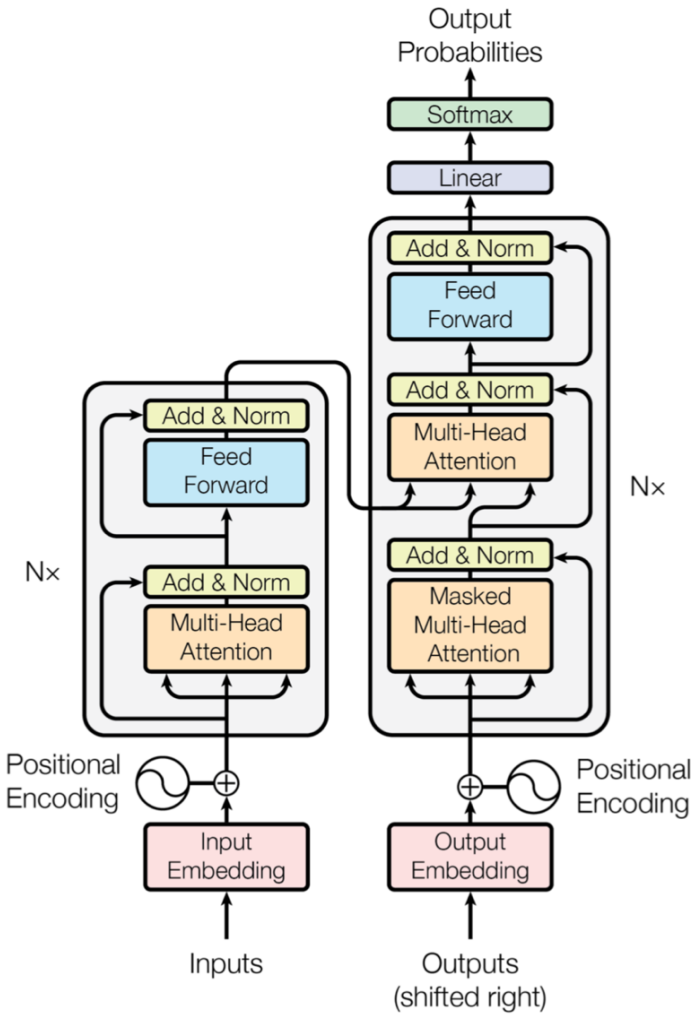

図 2.8. Transformer の構造

図の左側がエンコーダーであり，単語をインプットとして，文章の埋め込み表現を求める．この文章の埋め込み表現は，デコーダーにインプットする．そして，右側がデコーダーで，翻

訳語の単語列と文章の埋め込み表現をインプットとして，次の単語を予測する．

まずは，Positional Encoding を紹介する．Transformer は RNN を使わないので単語の順番の概念がない．Positional Encoding により単語の位置を考慮した埋め込み表現を作成する．式 2.7 と式 2.8 で Positional Encoding を計算し，単語の埋め込み表現に足す．

$$PE_{pos,2i} = \sin\left(pos/100000^{2i/d_{model}}\right) \tag{2.7}$$

$$PE_{pos,2i+1} = \sin\left(pos/100000^{2i/d_{model}}\right) \tag{2.8}$$

次は Attention のメカニズムを紹介する．Transformer では，Scaled Dot-Product Attention が使われている．Scaled Dot-Product Attention は図 2.9 で示す．

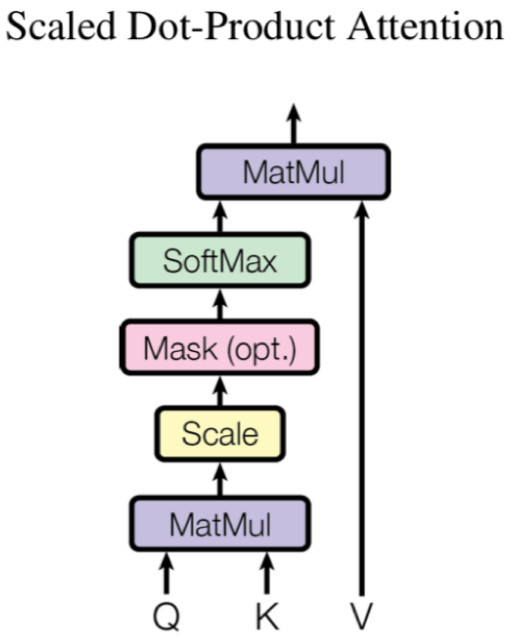

図 2.9. Scaled Dot-Product Attention

Attention のアウトプットは Query(Q) と Key(K) と Value(V) の 3 つのベクトルで計算される。各単語がそれぞれの Query と Key と Value のベクトルを持っている。より具体的には、Attention のアウトプットは V の加重和であり、その加重は Q と K を使って計算される。式 2.9 で示す．

$$Attention = softmax\left(\frac{QK^T}{\sqrt{d_{model}}}\right)V \tag{2.9}$$

QK^Tの要素の値が大きくなりすぎないように調整するために、Q と K の次元である d_{model} の平方根$\sqrt{d_{model}}$で割っている．

Multi-head Attention は上記の attention を複数並べることにより，精度の向上を図るものである．図 2.10 で示す．

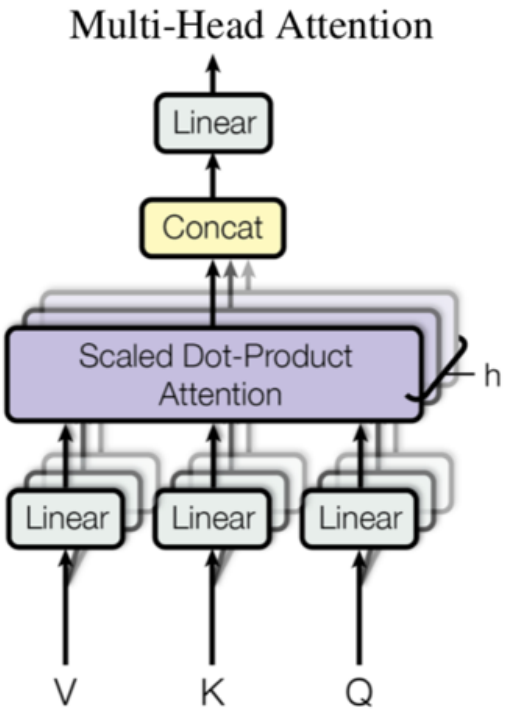

図 2.10. Multi-head Attention

Multi-head Attention の式は式 2.10 と式 2.11 で示す．

$$MultiHead(Q,K,V) = Concat\left(head_1,\cdots,head_h\right)W^O \tag{2.10}$$

$$where head_i = Attention\left(QW_i^Q, KW_i^K, VW_i^V\right) \tag{2.11}$$

Multi-head Attention レイヤの後に Feed Forward レイヤがある．式 2.12 で示す．

$$FFN(x) = \max\left(0, xW_1 + b_1\right)W_2 + b_2 \tag{2.12}$$

Feed Forward レイヤは，ReLU で活性化する $d_{ff} = 2048$ 次元の中間層と $d_{model} = 512$ 次元の出力層から成る 2 層の全結合ニューラルネットワークである．

2.2.4 評価指標

機械学習による分類問題でよく使われている評価指標を紹介する．コードクローンかどうかを分類することは二値分類問題に当たる．予測結果と実際結果の組み合わせは以下 4 つあります．

True Positive(TP)　Positive であると予測して，実際に Positive である
True Negative(TN)　Negative であると予測して，実際に Negative である
False Positive(FP)　Positive であると予測して，実際に Negative である
False Negative(FN)　Negative であると予測して，実際に Positive である

この 4 つの組み合わせを用いて，Accuracy, Precision, Recall, F1 score 4 つの評価指標を紹介する，

Accuracy とは Positive や Negative と予測したデータのうち，実際にそうであるものの割合．式 2.13 で示す．

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \tag{2.13}$$

Precision とは Positive と予測したデータのうち，実際に Positive であるものの割合．式 2.14 で示す．

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.14}$$

Recall とは実際に Positive であるもののうち，Positive であると予測されたものの割合．式 2.15 で示す．

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.15}$$

F1 score は Precision と Recall の調和平均である．式 2.16 で示す．

$$\text{F1} = \frac{2\text{Recall} \cdot \text{Precision}}{\text{Recall} + \text{Precision}} \tag{2.16}$$

2.3　cross-language コードクローンの関連研究

Cross-language コードクローンの検出は最近研究され始めている．今までの研究について簡単に紹介する．

Kraft *et al.*(2008)[7] は cross-language コードクローンの検出に関する最初の研究を行なった．Microsoft .NET フレームワークの CodeDOM ライブラリーを基づいて，C2D2 というツールを開発した．C#と VB.NET. の共通 CodeDOM グラフを生成できる．また, Al-omari *et al.*.(2012)[8] は共通中間言語（Common Intermediate Language (CIL)）を利用して C#, J#と VB.NET. のコードクローンを検出できる．二つとも.NET 言語についての研究である．Java と Python などの違うプラットフォームの言語には使えない．

X. Cheng *et al.*(2017)[9] は CLCMiner を開発した．多言語プロジェクトの Cross-language コードクローンは修正履歴 (diff) も類似するという仮説を基づいて研究を行なった．でも，修正履歴を持たない一般コードには使えない．

T. Vislavski *et al.*(2018)[10] は Budimac *et al.* の SSQSA アーキテクチャー [15] を利用して，コードを enriched Concrete Syntax Tree (eCST) に変換して行列を生成する．それを用いてコードクローンを検出する．しかし，比較できるコードは制限がある．コードの長さ，ステップとフローが同じでなければならない．実用的に使うには難しい．

Perez *et al.*(2019)[11] は AST を利用してトークンの分散表現を生成する．それから，競技プログラミングのデータを教師データとして，教師あり学習でコードクローンを検出する，K.Nafi *et al.*(2019)[12] はあらかじめ決めた文法特徴を使って，コードを分析する．それから，API ドキュメントの類似度を利用したフィルタを使う．最後に，on-the-fly または競技プログラミングのデータを使った教師あり学習を行う．Perez *et al.* と同じく，競技プログラミングの教師データは良いデータではないため，精度は高くない．

2.4 教師あり学習による検出の問題点

Cross-language コードクローンの検出は研究され始めている．そして，昨年に，教師データを使って，教師あり学習で cross-language コードクローンを検出する論文が二つ出た [11, 12] が，あまり性能が良くない．論文は二つとも教師データとして，AtCoder[*1]や Google CodeJam[*2]などの競技プログラミングのデータを使って，学習した．競技プログラミングのコードはコーディングスタイルや解決方法に差が大きすぎるため，良いデータではない．教師あり学習はデータセットの質に依存するため，良いデータセットが存在しない今の状況では，まだ性能が高くならない．そして，教師あり学習は学習されていない他の領域のパターンに弱い．競技プログラミングで学習されたモデルは，競技プログラミングのコードに使えるが，実用的の領域には使えない．実用的の領域にはまだ学習データが存在していないため，実応用にはまだ現実的ではない．

[*1] `https://atcoder.jp`

[*2] `https://www.go-hero.net/jam/10/languages/0`

第 3 章

教師なし学習による cross-language コードクローン検出

3.1 教師なし学習による検出の利点

教師ラベルなし単言語学習データのみを使った教師なし学習による cross-language コードクローン検出を提案する．

教師なし学習による cross-language コードクローン検出の基本的考えは，教師データ不要のあるタスクをモデルに学習させて，その副産物としてコードの分散表現が得られる．コードの分散表現のコサイン類似度を計算して，コードクローンを判断する．

教師なし学習は教師あり学習に比べて，主に以下三つの利点が期待できる：

1) 性能に期待できる．教師あり学習は教師データの質に依存するため，まだ良い教師データが存在しない今の状況では，性能は高くない．一方，教師ラベルなしの単言語データはたくさん存在する．教師なし学習の精度が期待できます．
2) 汎用性に期待できる．教師あり学習は学習されたパターンに強く，学習されていない他の領域のパターンに弱い．今までの論文は競技プログラミングの教師データを使って学習した．実用的の領域にはまだ学習データが存在していないため，実用に応用することはできない．一方，教師ラベルなしの単言語データはどの領域でもたくさん存在している．実用にすぐ応用できる．
3) モデルに通す計算量が低い，大規模の検出にも応用できる．教師あり学習はコードクローンのペアをインプットとして，クローンかどうかをアウトプットする．検出候補のコードを二つずつ組み合わせてモデルに通すため，モデルに通す計算量は $\mathrm{O}(n^2)$．教師なし学習はコードを一つずつモデルに通して，コードの分散表現をアウトプットする．後は分散表現のコサイン類似度を計算して，コードクローンかどうかを判断する．モデルに通す計算量は $\mathrm{O}(n)$ である．モデルに通す時間コストはコサイン類似度を計算するより遥かに大きいため，モデルに通す計算量を抑えると，検出全体の時間は大幅に抑えられる．大規模のコードクローン検出にも応用できる．

教師なし学習による cross-language コードクローン検出は我々が知る限り，まだ研究されたことのないタスクであるため，その可能性を追求するに，二つの仮説を立てた．

3.2 仮説 1

自然言語と違い，プログラミング言語は違う言語でもたくさんの共通点があり，これらの共通点をもとに，言語の間の違う部分もベクトル空間上の近い位置に学習できる．

自然言語における cross-language タスクは，主に二つの言語を関連付けた教師データを使い，教師あり学習を行う．例えば，言語翻訳タスクは，事前に与えられた二つの言語の翻訳文ペアを学習データとして使い，そうすることで二つの言語を関連付ける．自然言語では，言語によって使う文字や文法は全然違うため，教師データをなしに二つの言語を関連付けるのは難しい．だが，プログラミング言語はそうではない．プログラミング言語は違う言語でもたくさんの共通点がある．特に，汎用的な言語は共通点が多い．

共通点の例として：

- 入れ子構造がある
- 式を使ってデータ処理をする
- if, for, while, function, class などの statement がある
- 英語ベースで変数名や関数名を付ける
- AST に変換して，実行する

教師なし学習では，トークンの意味をベクトル化する．言語の間の共通点をもとに，違う言語のベクトル空間は独立なものでは無くなって，関連付けられる．言語の間の違い部分もベクトル空間上の近い位置に学習できる．

3.3 仮説 2

プログラミング言語は分布仮説に従っていて，英語ベースで付けられた変数名や関数名は隠しラベルの役目を果たす．

分布仮説とは，同じ文脈で出現する単語は意味も類似する [16]．言い換えれば，単語の意味は周辺文脈によって定義される [17]．分布仮説は統計的意味論 [18] の基礎である．

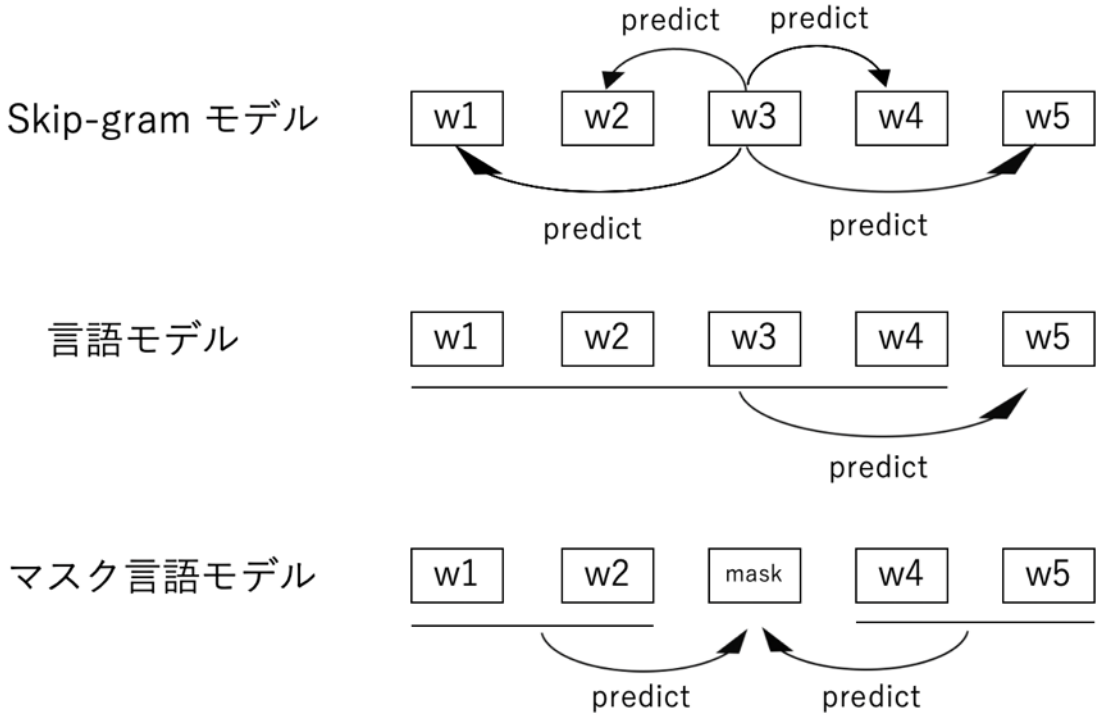

図 3.1. 教師なし学習手法

分布仮説は近年発展している教師なし学習手法と強く関係していると考える．図 3.1 に示した通り，例えば，skip-gram モデル [19] は，単語をインプットとして，その単語の周りの単語を予測する．言語モデルは，今までの文脈をインプットとして，次の単語を予測する．マスク言語モデル [20] は，マスクされていない単語をインプットとして，マスクされた単語を予測する．

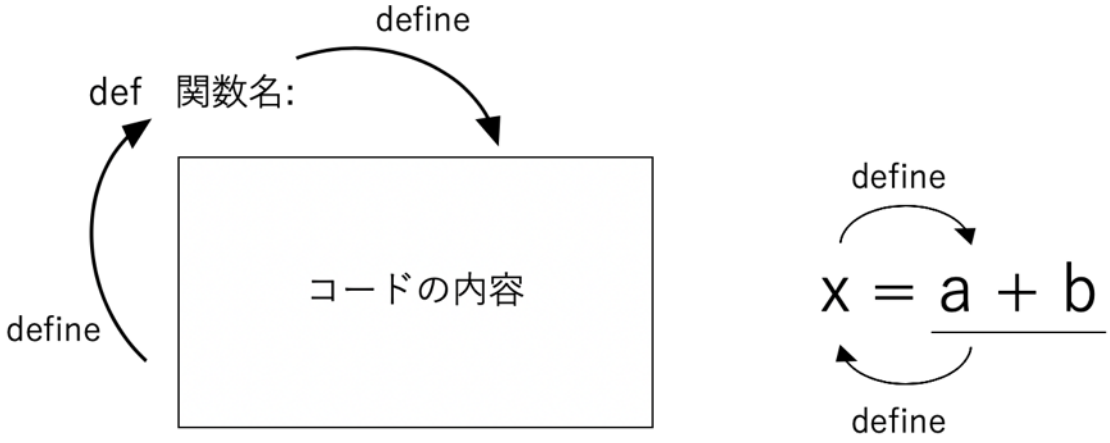

図 3.2. プログラミング言語の形

プログラミング言語は，形から見ると，分布仮説に従っていると思われる．図 3.2 に示した通り，例えば，関数を定義する時，関数名を書いてから，後ろにコードの内容を書く．関数名

の意味は後ろのコードによって定義される．そして，後ろのコードの意味も関数名によって定義される．式の場合も同じ．例えば，x = a + b の場合は，x の意味は a + b によって定義される．a + b の意味もまた x によって定義される．

違う言語でも，同じく英語ベースで関数名や変数名を付けている．そして，機能的に近いコードは，関数名や変数名も関係している．分布仮説が成立すれば，変数名や関数名は隠しラベルの役目を果たす．これで，教師なし学習が可能になる．

第 4 章

手法と実験結果

LSTM[13] を使った autoencoder[21] や Transformer[14] を使った言語モデルとそれぞれの改善法合計五つの手法について実験した．4.1 は学習データ，評価データ，データの前処理について紹介する．4.2 から 4.6 まではそれぞれの手法と実験結果について紹介する．4.7 は教師あり学習手法と同じ評価データで比較する．4.8 は実験結果について議論する．

4.1 データセットと前処理

4.1.1 学習データ

学習データは GitHub から集めた．GitHub から LeetCode や algorithm のキーワードを含めたリポジトリを Java と Python それぞれ 550 個を集めた．長さ 512 トークン以上のファイルを取り除いて，Java は 123623 ファイル，Python は 83398 ファイルを学習データとして使用した．

LeetCode[*1] は IT 企業のコーディング面接を練習するためのサイトである．LeetCode を学習データとして選んだ理由は，競技プログラミングと違って，LeetCode は時間を競うではなく，コーディングスタイルを重視している．LeetCode は競技プログラミングより，実際の仕事現場で書いたコードに近いである．

4.1.2 評価データ

評価データは LeetCode の解答を使う．同じ問題を解く Java と Python のコードは cross-language コードクローンと見なす．103 個の問題にそれぞれ一つ Java の解答と一つ Python の解答がある．組み合わせて，全部 10609 ペアの中に 103 ペアだけの正解ペアを検出するというタスクになる．実際の実用の時も，コードクローンの数に対してコードの種類数が遥かに多いである．そのため，たくさん種類のコードの中に，数少ないコードクローンを見つかられるかどうかを評価する．

[*1] https://leetcode.com

4.1.3 BPE(byte pair encoding)

BPE(byte pair encoding)[22] はデータ圧縮法の一つである．自然言語では，BPE を使って，非頻出語をサブワードに分けることで未知語を対応することがある．

BPE は特にプログラミング言語に適していると考える．プログラミング言語は開発者が自分で名前を付けるため，未知語は避けられない．今までの研究は非頻出語を identifier などの型に変えることが多い．しかし，そうすると多くの情報が損失することになる．プログラミング言語で名前を付ける時よくキャメルケースやスネークケースを使って意味を表現する．BPE を使えば，上手く名前を意味のあるサブワードに分けられるではないかと思って，実験をした．結果は，ほとんどの変数名や関数名は意味のあるサブワードに分けることができた．

学習データに BPE を使うことにした．Java と Python のコードの語彙数は合わせて 157347 である．BPE を使って，30829 にした．

4.2 手法 1

図 4.1. 手法 1

LSTM[13] を使った autoencoder[21] について実験した．図 4.1 で示した通り，コードを LSTM encoder に入れて，一つの code embedding にエンコードする．そして，LSTM decoder は code embedding を元に戻すように学習する．学習の結果として，コードをコードの意味を含めた一つの code embedding にエンコードできる encoder を得られる．評価するとき，その encoder を使って，コードをエンコードする．得られた code embedding のコサイン類似度を計算して，コードクローンかどうかを判断する．レイヤーは一つで，embedding は次元数は 512 である．予想として，二つの言語を一つのモデルで学習することで，仮説 1 で言語の間の共通点をもとに，言語の間の違い部分もベクトル空間上の近い位置に学習できる．

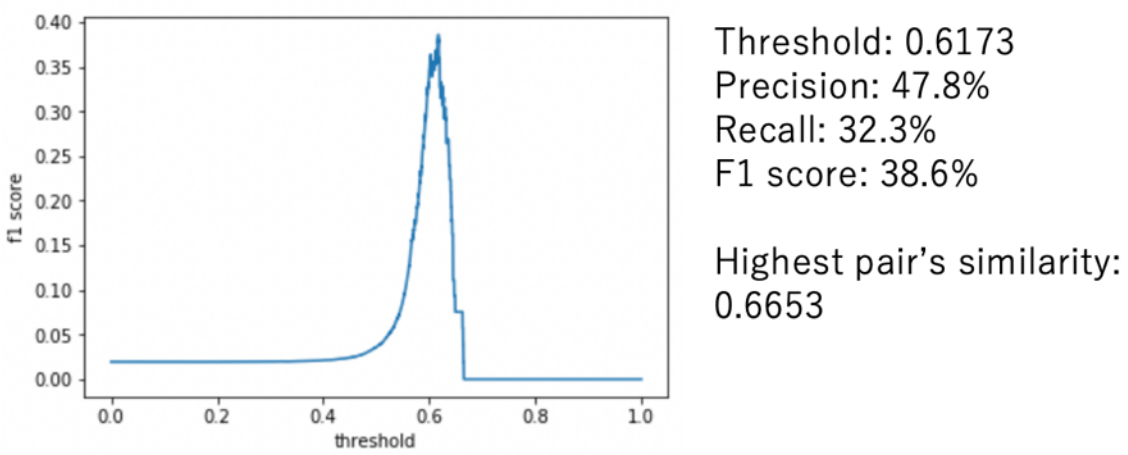

図 4.2. 手法 1 の結果

結果は図 4.2 で示した．F1 は 38.6% である．10609 ペアの中に 103 ペアが正解なので，ランダムで判断する場合は正解率約 1% である．したがって，性能は確かにあった．しかし，高いとは言えない．そして，一番近いペアの類似度は 0.66 なので，Java のコードと Python のコードの距離が離れていることが分かった．二つの言語の間の違いが code embedding に含まれていると考えられる．

4.3　手法 2

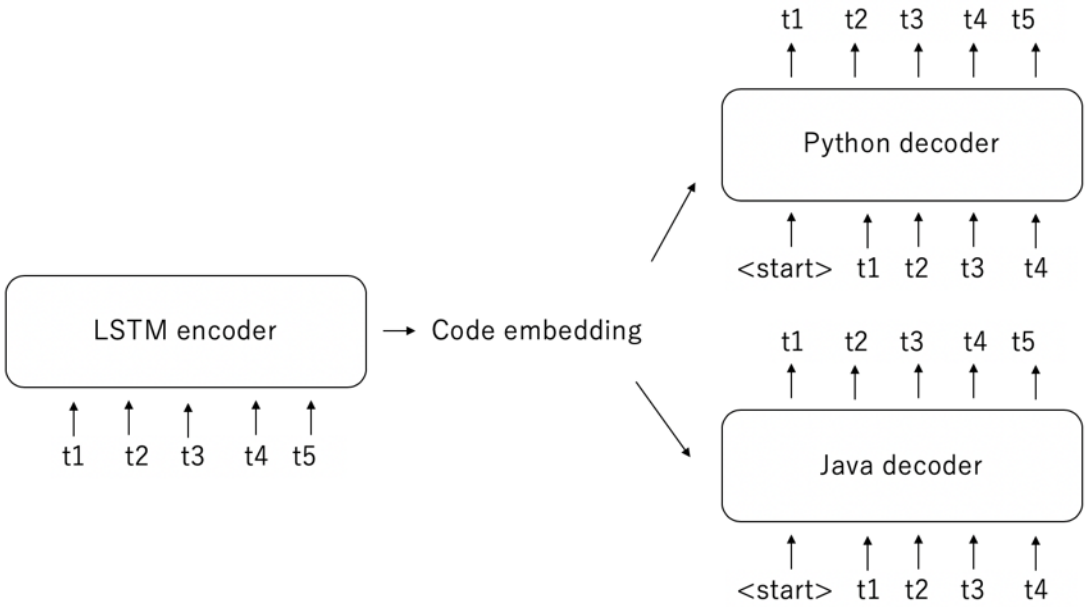

図 4.3. 手法 2

図 4.3 で示した通り，言語ごとに違う decoder を使用することにした．二つの言語のコードは同じ encoder でエンコードするが，元に戻す時にそれぞれの decoder を使う．手法 1 で二つの言語の間の違いが code embedding に含まれていることが分かった．二つの decoder を使うことで，言語の間の違い部分が decoder で吸収され，意味に関わる部分だけ code embedding に記憶させる．

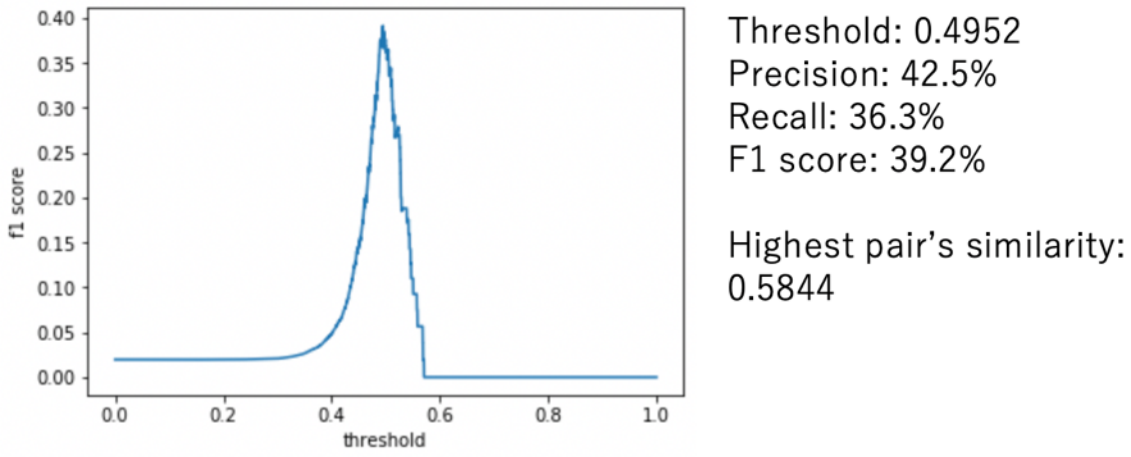

図 4.4. 手法 2 の結果

結果は図 4.4 で示した．F1 は 39.2% でちょっと上がったが，一番近いペアの類似度は 0.58 で逆に下がりました．二つの decoder を使うことは言語の間の距離を縮めないことが分かった．

4.4 手法 3

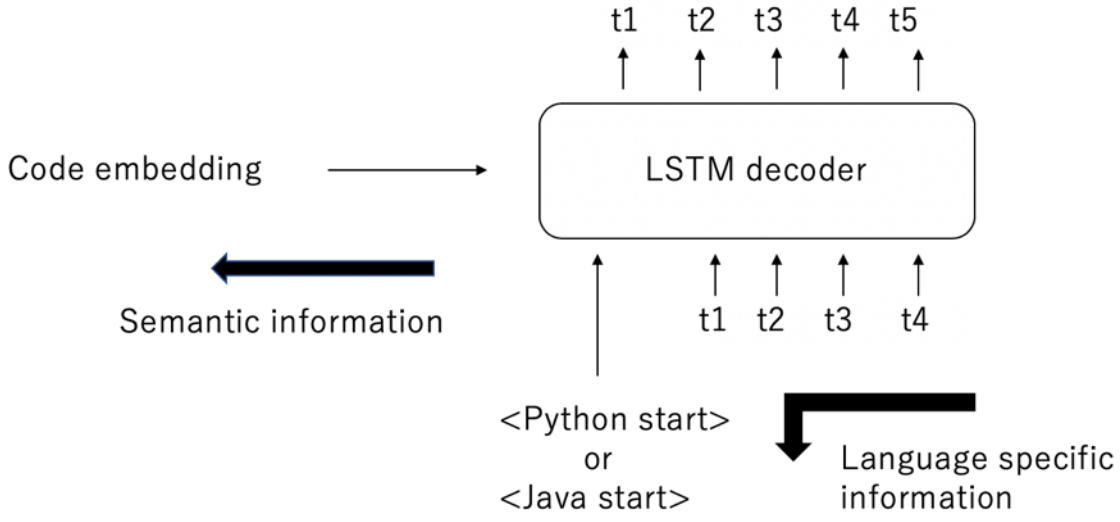

図 4.5. 手法 3

図 4.5 で示した通り，言語ごとに違うスタートトークンを使うことにした．手法 2 で，二つの decoder を使うことは言語の間の距離を縮めないことが分かったので，言語の間の違いを decoder に記憶させるではなく，スタートトークンの embedding に記憶させる．decoder は二つのインプットがある．一つは code embedding で，一つはスタートトークンの word embedding である．スタートトークンの embedding は code embedding と同じ記憶空間がある．言語ごとに違うスタートトークンを使うことで，言語に関する情報はスタートトークンの embedding に記憶させ，意味に関する情報は code embedding に流れさせる．

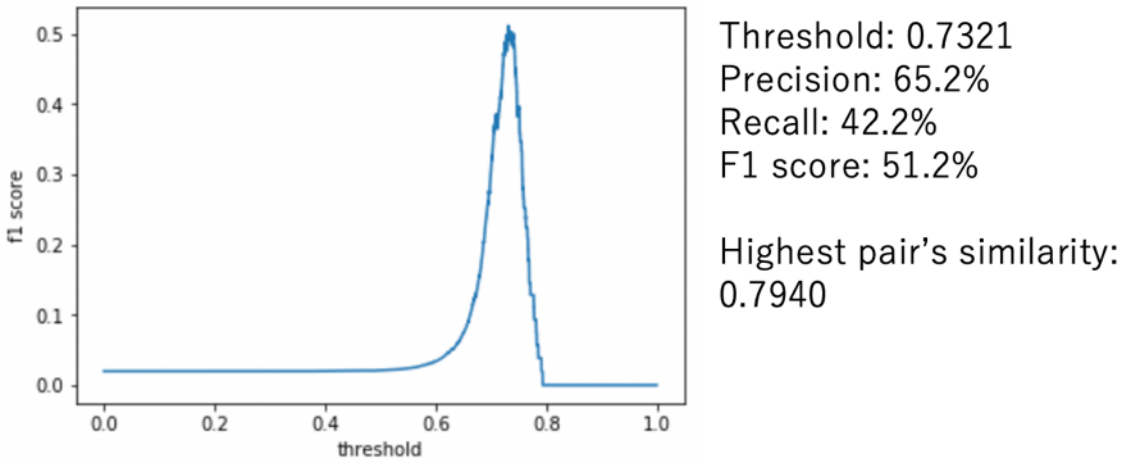

図 4.6. 手法 3 の結果

結果は図 4.6 で示した．F1 は 51.2% で，大幅に上がりました．そして，一番近いペアの類似度は 0.79 まで上がった．予想通りに，言語ごとに違うスタートトークンを使うことで，言語の間の距離を縮めることができた．

4.5 手法 4

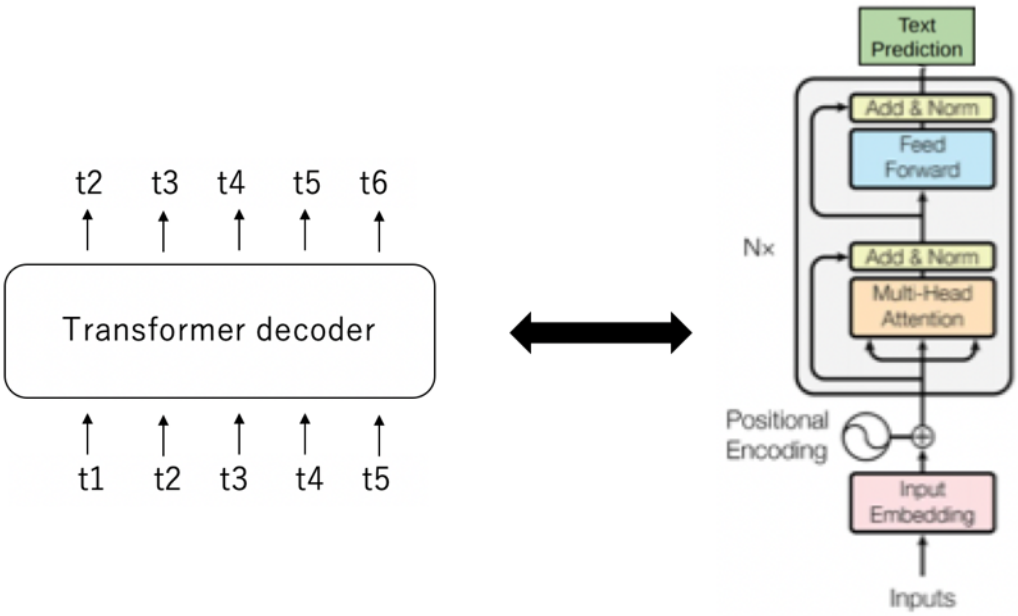

図 4.7. 手法 4

Transformer[14] を使った language model について実験をした．language model とは前の文脈を見て，次の単語を予測し続けるモデルである．モデルは Transformer の decoder だけを使った．図 4.7 で示した．レイヤーは 6 で，次元数は 512 である．評価する時に，後ろの padding を取り除いたアウトプットを各トークンの平均を取ってコードの分散表現にする．得られたコードの分散表現のコサイン類似度を計算して，コードクローンかどうかを判断する．予想として，二つの言語を一つのモデルで学習することで，仮説 1 で言語の間の共通点をもとに，言語の間の違い部分もベクトル空間上の近い位置に学習できる．そして，language model を使うことで，仮説 2 で分布仮説が成立するのおかげで，変数名や関数名は隠しラベルの役目を果たす．

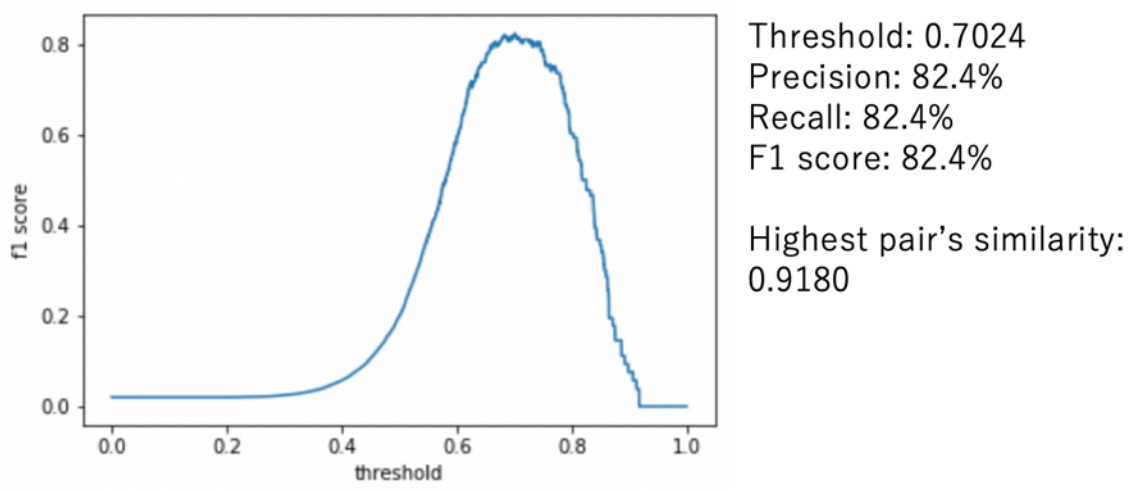

図 4.8. 手法 4 の結果

結果は図 4.8 で示した．F1 は 82.4% でかなり高い性能が出た．しかも一番近いペアの類似度は 0.918 である．二つの仮説で予想した通り，違う言語のコードでも距離がかなり近くなった．

4.6　手法 5

手法 4 で得られたコードの分散表現の 512 次元の中に言語に関わる特徴と意味に関わる特徴があると予想した．言語に関わる特徴の影響を減らせる実験をした．

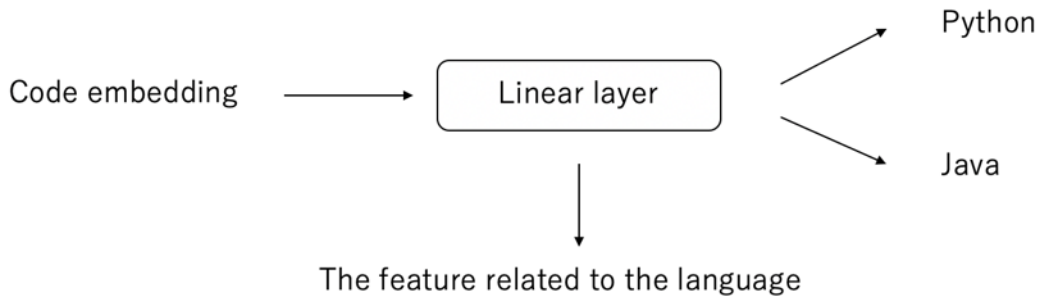

図 4.9. 手法 5

図 4.9 で示した通り，コードの分散表現をインプットとして，一つの linear 層の classifier を使って，Python か Java かと分類させる．その linear 層の重みの絶対値が高いところは言語に関係強い特徴であると予想した．学習時に L1 正則化をかけた．L1 正則化をかけることで，予測に影響が弱い特徴の重みは 0 に近く，予測に影響が強い特徴だけ重みが高く学習できる．

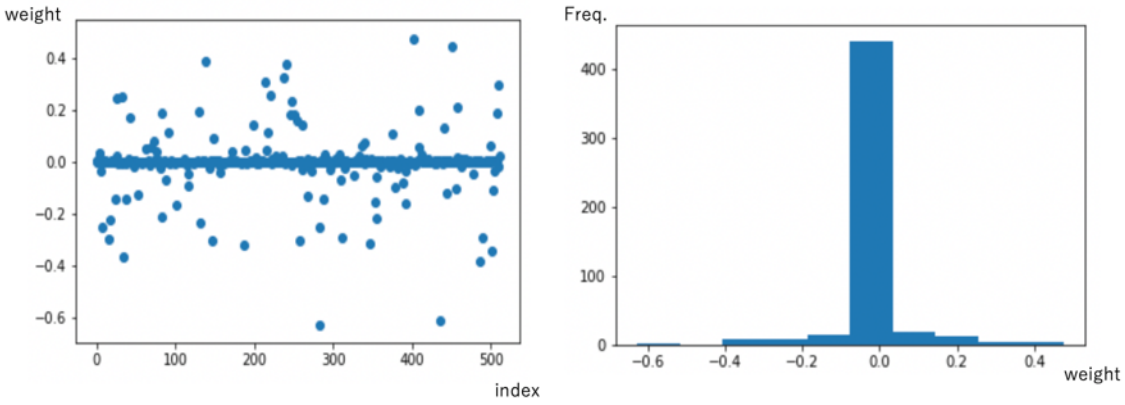

図 4.10. linear 層の重み

結果は精度 99.97% で分類できる．重みは図 4.10 に示した通り，ほとんどの重みは 0 に近いである．残りの重みの絶対値が高い特徴は言語に関わる特徴だと思われる．

重みの絶対値が高い特徴は以下の式で減らせる．

$$v_i' = (1 - \min(\mid w_i \mid + \alpha, 1)) \cdot v_i \tag{4.1}$$

式の中，v_i はコードの分散表現にインデックスが i である要素．v_i' は減らした後の要素である．w_i は linear 層の重みにインデックスが i である要素．α はハイパーパラメーターである．重みの絶対値が $1 - \alpha$ 以上の特徴は取り除かれる．残りの特徴は重みの絶対値が大きければ大きいほど減らされる．結果として，α が 0.95 の時一番高い性能が得られた．

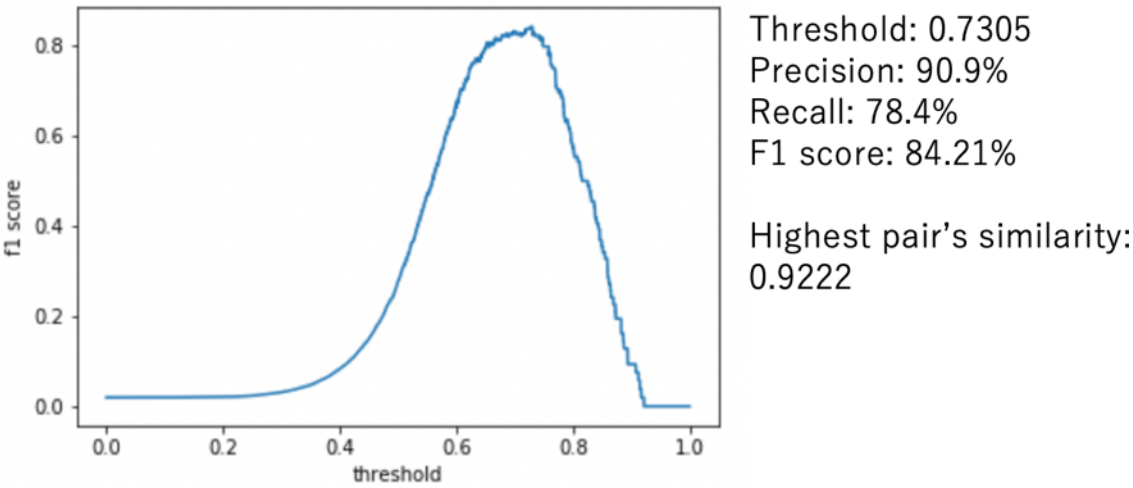

図 4.11. 手法 5 の結果

結果は図 4.11 で示した．F1 は 84.2% で，性能がちょっと上がった．そして，一番近いペアの類似度は 0.922 で，もっと近くなった．性能はちょっとしか上がらなかったのはデータセットに原因があると思われる．4.8 で議論する．

表 4.1. AST ノードの分散表現学習のハイパーパラメータ

パラメータ	値
Ancestors window size	2
Descendants window size	1
Siblings included	No
Output vector dimension	100

表 4.2. 教師あり学習のハイパーパラメータ

パラメータ	値
Token vector dimension	100
Encoder layer	bidirectional LSTM, stacked with 2 layers
layer dimensions	100 and 50
Classifier	single hidden layer, 64 units
Optimizer	RMSprop
Epochs	10

4.7　教師あり学習手法との比較

教師あり学習手法である Perez *et al.*(2019)[11] が提案した手法と同じ評価データで比較した．教師ラベル付きの学習データは Perez *et al.* の論文と同じく競技プログラミングのデータを使った．そして，評価データは本研究と同じく Leetcode の評価データを使った．競技プログラミングのコードと Leetcode のコードは両方ともアルゴリズムの問題を解決するコードである．教師あり学習手法で学習されたモデルは同じドメインのコードに汎用性があるかどうかを問われる．実応用にも，教師ラベルありのデータは無限に作られる訳にはいかない．少なくとも同じドメインへ汎用性が必要と思われる．

実験の詳細を紹介する．Perez *et al.* の手法は 2 つの手順に分けれる．1 つ目は教師なし学習で AST ノードの分散表現を得る．2 つ目は学習された分散表現を使って LSTM ベースの教師あり学習でコードクローン検出の学習をする．

手順 1 の教師なし学習は GitHub のプロジェクトを学習データとして使われている．GitHub から Jave のプロジェクト 1027 個，Python のプロジェクト 879 個を集めた．AST に変換して，表 4.1 で示したハイパーパラメータで教師なし学習を行った．語彙数は 10000 にした．頻出したトークンは value で学習させて，語彙数を超えた非頻出語は型で学習させた．

手順 2 は競技プログラミングのデータで学習した．同じ問題を解く Java と Python のコードはコードクローンと見なす．データセットから Positive ペアと Negative ペアを組み合わせて，教師あり学習を行った．学習の時のハイパーパラメータは表 4.2 で示した．

表 4.3. Perez*et al.* の論文にあった評価結果

評価データに Positive ペアと Negative ペアの割合	F1-score
学習時と同じく 1 対 4	0.66
All-combination	0.32

表 4.4. 学習時に違う割合と評価時に違う判定方法の F1-score

評価時の判定方法 学習時の割合	round 関数法の F1	ベスト閾値法の F1
1:1	0.027	0.045
1:4	0.040	0.077
1:10	0.047	**0.139**
1:20	0.040	0.067
1:100	0	0.041

評価データは本研究の手法と同じく LeetCode のデータセットを使った．Perez *et al.* の論文にも書いてあり，評価データの Positive ペアと Negative ペアの割合によって，結果は大きく変わっている．表 4.3 は Perez *et al.* の論文に書いてあった評価データは学習データと同じく競技プログラミングデータを使った評価結果である．評価する時，Positive ペアと Negative ペアの割合は学習の時と同じく 1 対 4 にした時，F1 は 0.66 に達したが，実応用と同じく，検出候補のコードの all-combination で評価した場合は，F1 は 0.32 に下がった．本研究評価データは実応用の同じく all-combination を採用した．103 個の Python コードと 103 個 Java コードを組み合わせて，全部 10609 ペアの中に 103 ペアの正解ペアがある．

教師あり学習であるため，学習する時に，Positive ペアと Negative ペアの割合も実験結果に大きく影響することは実験でわかった．学習する時，Positive ペアと Negative ペアの割合を 1:1，1:4，1:10，1:20 と 1:100 五つのパターンで実験した．実験結果は表 4.4 で示す．評価する時にコードクローンの判定は二つの方法を使った．一つ目は round 関数を使った方法である．学習する時に round 関数を使って，0.5 以上はコードクローンで，0.5 以下はコードクローンじゃないように学習した．しかし，評価する時は，学習時と同じく round 関数を使うと，性能が完全に発揮することができないことがわかった．だから，閾値を使った評価方法を使った，0.5 を基準ではなく，一番高い性能を出せる閾値を探し出す．図 4.12 は Positive ペアと Negative ペアの割合が 1:10 の時に，閾値と F1-score の関係を示した．そして，表 4.5 は割合が 1:10 の時の詳細結果を示した．

表 4.5. 割合が 1:10 の時の詳細結果

評価時の判定方法	round 関数法	ベスト閾値法
Best threshold	0.5	0.832
Precision	0.025	0.144
Recall	0.667	0.133
F1-score	0.047	0.139

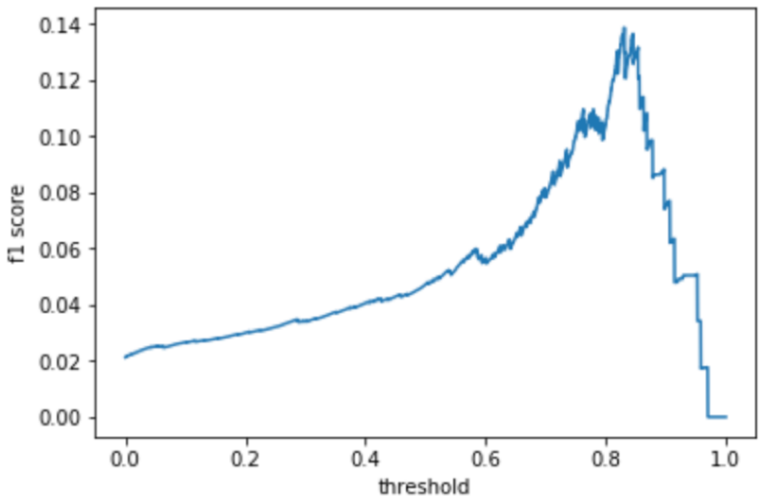

図 4.12. Positive ペアと Negative ペアの割合が 1:10 の時に閾値と F1-score の関係

実験の結果，学習の時に Positive ペアと Negative ペアの割合が 1:10 の時，そして，評価する時の判定法はベスト閾値法の時に一番高い F1-score を得られた．一番高い F1-score は 0.139 であった．本研究の五つの手法と比較して，結果は表 4.6 でまとめた．

表 4.6. 教師あり学習手法と本研究の五つの手法の比較

	Precision	Recall	F1-score	計算量	評価時間
教師あり学習手法	14.4%	13.3%	13.9%	$O(n^2)$	486s
手法 1	47.8%	32.3%	38.6%	$O(n)$	1.22s
手法 2	42.5%	36.3%	39.2%	$O(n)$	1.28s
手法 3	65.2%	42.2%	51.2%	$O(n)$	1.26s
手法 4	82.4%	82.4%	82.4%	$O(n)$	2.13s
手法 5	90.9%	78.4%	84.2%	$O(n)$	2.17s

結果は提案した全ての手法は教師あり手法より大幅に上回った．教師あり学習手法の F1 の 13.9% に対して，一番性能が高い手法 5 では 84.2% の F1 を得られた．そして，計算時間も大

幅に抑えた．教師あり学習手法は AST に構文解析する必要がある．そして，モデルに通す計算量は $O(n^2)$ である．教師なし学習手法は AST に変換する必要がない．そして，モデルに通す計算量は $O(n)$ である．実際の評価時間では，10609 ペアを評価するのに，教師あり学習手法は 8 分以上が掛かったに対して，提案した教師なし学習手法は 1 秒 2 秒に抑えることができた．

4.8 議論

教師あり学習手法の F1 は 13.9% に対して，手法 5 では，F1 は 84.2% を得られた．かなり高い精度と思われる．性能の数字だけじゃなく、もっと正確にモデルの性能を評価するため，モデルが正解だと判断したペアの中に，間違ったペアを見てみた．

図 4.13 から図 4.16 まではコードペアのサンプルが挙げられた．左は Python のコードで，右は Java のコードである．こちらのペアはモデルが近いと判定されたが、違う問題を解くコードなので、間違いだとカウントされた。でも、これらのペアは違う問題のコードだけど、実はかなり近いであった。

```python
class Solution(object):
    def fixedPoint(self, A):
        l, h = 0, len(A) - 1
        while l <= h:
            mid = (l + h) // 2
            if A[mid] < mid:
                l = mid + 1
            elif A[mid] > mid:
                h = mid - 1
            else:
                return mid
        return -1
```

```java
class Solution {
    public int peakIndexInMountainArray(int[] A) {
        int lo = 0, hi = A.length - 1;
        while (lo < hi) {
            int mid = (lo + hi) / 2;
            if (A[mid] < A[mid + 1]) lo = mid + 1;
             else hi = mid;
        }
        return lo;
    }
}
```

図 4.13. サンプル 1

例えば，図 4.13 サンプル 1 のペアはどっちでも binary search のアルゴリズムを使ったコードである．

```python
class Solution(object):
    def topKFrequent(self, words, k):
        count =  collections.Counter(words)
        heap = [(-freq, word) for word, freq in count.items()]
        heapq.heapify(heap)
        return [heapq.heappop(heap)[1] for _ in xrange(k)]
```

Python:
Problem 692:
Top K Frequent Words

Java:
Problem 347:
Top K Frequent Elements

```java
class Solution {
    public List<Integer> topKFrequent(int[] nums, int k) {
        HashMap<Integer, Integer> count = new HashMap();
        for (int n: nums) {
            count.put(n, count.getOrDefault(n, 0) + 1);
        }
        PriorityQueue<Integer> heap =
new  PriorityQueue<Integer>((n1, n2) -> count.get(n1) - count.get(n2));
        for (int n: count.keySet()) {
            heap.add(n);
            if (heap.size() > k)
                heap.poll();
        }
        List<Integer> top_k = new LinkedList();
        while (!heap.isEmpty())
            top_k.add(heap.poll());
        Collections.reverse(top_k);
        return top_k;
    }
}
```

図 4.14. サンプル 2

図 4.14 サンプル 2 のペアは，問題が違うけど、実は同じようなことをやっている。コードクローンであると考えられる．モデルはうまくこれらのコードをコードクローンだと認識している．

```python
class Solution(object):
    def findDisappearedNumbers(self, nums):
        res = []
        if nums:
            n = len(nums)
            for i in range(n):
                val = abs(nums[i]) - 1
                if nums[val] > 0:
                    nums[val] = -nums[val]
            for i in range(n):
                if nums[i] > 0:
                    res.append(i + 1)
        return res
```

Python:
Problem 448:
Find All Numbers Disappeared in an Array
Input: [4,3,2,7,8,2,3,1]
Output: [5,6]

```java
class Solution {
    public int missingNumber(int[] nums) {
        int res = nums.length;
        for (int i = 0; i < nums.length; i++) {
            res ^= i;
            res ^= nums[i];
        }
        return res; }
}
```

Java:
Problem 268:
Missing Number
Input: [9,6,4,2,3,5,7,0,1]
Output: 8

図 4.15. サンプル 3

図 4.15 サンプル 3 のペアはちょっと興味深い．完全に違うアルゴリズムを使っている。でも、実は目的が似ていて、一つは 1 から n まで全て無くなって数字を探し出しこと、一つは 1 から n まで一つだけ無くなった数字を探し出すことである。アルゴリズムが違うし、そして、関数名も違う言い方や単語をつかっている。なぜか、モデルはコードクローンだと認識している．

```python
class Solution(object):
    def removeNthFromEnd(self, head, n):
        if head is None:
            return None
        slow = fast = head
        for i in range(n):
            fast = fast.next
        if fast is None:
            head = head.next
            return head
        while fast.next is not None:
            fast = fast.next
            slow = slow.next
        curr = slow.next
        slow.next = curr.next
        return head
```

```java
class Solution {
    public ListNode middleNode(ListNode head) {
        ListNode fast, slow;
        fast = slow = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
    }
}
```

図 4.16. サンプル 4

図 4.16 サンプル 4 のペアははどっちでも 2 pointer のテクニックを使っている。一つは LinkedList の真ん中のノードを探し出すこと、一つは後ろから n 番目のノードを見つけて削除する。アルゴリズム的にはかなり近いである。

サンプル以外の他のペアもほとんどは違う問題だけど、アルゴリズムあるいは目的が近いコードである．これらのペアは場合によって，価値のあるコードクローンだと考えられる。

第 5 章

まとめと今後の課題

5.1 まとめ

本研究では二つの仮説を元に，教師なし学習での cross-language コードクローン検出を試みた．Cross-language コードクローンの検出は研究され始めている [7, 8, 9, 10, 11, 12]．そして，昨年に，教師データを使って，教師あり学習で cross-language コードクローンを検出する論文が二つ出た [11, 12] が，あまり性能が良くない．論文は二つとも教師データとして競技プログラミングのデータを使っていた．競技プログラミングのコードはコーディングスタイルや解決方法に差が大きすぎるため，良いデータではない．教師あり学習はデータセットの質に依存するため，良いデータセットが存在しない今の状況では，実応用にはまだ現実的ではない．

そこで，本研究は教師データが必要ない教師なし学習での cross-language コードクローンの検出を試みた．教師なし学習は教師あり学習に比べて，主に以下三つの利点が期待できる：

1） **性能に期待できる．**
2） **汎用性に期待できる．**
3） **モデルに通す計算量が低い，大規模の検出にも応用できる．**

教師なし学習による cross-language コードクローン検出は我々が知る限り，まだ研究されたことのないタスクであるため，その可能性を追求するに，以下二つの仮説を立てた．

仮説 1 自然言語と違い，プログラミング言語は違う言語でもたくさんの共通点があり，これらの共通点をもとに，言語の間の違う部分もベクトル空間上の近い位置に学習できる．

仮説 2 プログラミング言語は分布仮説に従っていて，英語ベースで付けられた変数名や関数名は隠しラベルの役目を果たす．

二つの仮説を元に，LSTM を使った autoencoder や Transformer を使った言語モデルとそれぞれの改善法合計五つの手法を提案し，実験した．学習データとして GitHub からアルゴリズムに関係する Java と Python のコードを集め，評価は LeetCode の解答を使って行った．また，教師あり学習手法と比較した．

表 5.1. 教師あり学習手法と本研究の五つの手法の比較

	Precision	Recall	F1-score	計算量	評価時間
教師あり学習手法	14.4%	13.3%	13.9%	$O(n^2)$	486s
手法 1	47.8%	32.3%	38.6%	$O(n)$	1.22s
手法 2	42.5%	36.3%	39.2%	$O(n)$	1.28s
手法 3	65.2%	42.2%	51.2%	$O(n)$	1.26s
手法 4	82.4%	82.4%	82.4%	$O(n)$	2.13s
手法 5	90.9%	78.4%	84.2%	$O(n)$	2.17s

実験の結果は表 5.1 でまとめた．結果は提案した全ての手法は教師あり手法より大幅に上回った．教師あり学習手法の F1 の 13.9% に対して，一番性能が高い手法 5 では 84.2% の F1 を得られた．そして，計算時間も大幅に抑えた．AST に構文解析する必要がなくなり，モデルに通す計算量は教師あり学習手法の $O(n^2)$ から $O(n)$ に抑えた．実際の評価時間では，10609 ペアを評価するのに，教師あり学習手法は 8 分以上が掛かったに対して，提案した教師なし学習手法は 1 秒 2 秒に抑えることができた．

そして，性能の数字だけじゃなく、もっと正確にモデルの性能を評価するため，モデルが正解だと判断したペアの中に，間違ったペアを見てみた．これらのペアのほとんどは違う問題だけど、アルゴリズムあるいは目的が近いコードである．これらのペアは場合によって，価値のあるコードクローンだと考えられる。従って，提案した手法で学習したモデルは cross-language コードクローンを検出ことができた．結果から見ると，提案した二つの仮説が成立すると考えられる．

5.2 今後の課題

本研究では二つの仮説を元に，教師なし学習での cross-language コードクローン検出を試みた．今後の課題としては，三つの方向について考えた．

一つ目は，性能がさらに上がる手法を探し出すことである．本研究では，いくつかの教師なし学習手法を提案して，高い性能を収めた．今後の課題として，更なる性能が高い手法を実験して，探し出すこと．例えば，マスク言語モデルなどの他の教師なし学習手法や DANN(Domain Adversarial Neural Networks) などの教師なしドメイン適応手法について実験する．

二つ目は，二つの仮説はどこまで適応するかを更なる検証することである．本研究は Java と Python 二つの代表的言語について実験して，高い性能から二つの仮説が成立することを示した．他の言語に適応するために，他の言語の組み合わせについて実験する必要がある．二つの仮説はどこまで適応するかを検証するべきだ．

三つ目は，実応用のプロジェクトで評価する．本研究は LeetCode のデータセットを使って，アルゴリズムを解くコードについて実験した．アルゴリズムを解くコードだけではなく，

実際のプロジェクトの中に，本当の cross-language コードクローンを検出できるかどうかを更なる実験する必要がある．

発表文献と研究活動

(1) 劉宇澤，千葉滋．教師ラベルなし単言語学習データのみでの cross-language コードクローン検出の試み．日本ソフトウェア科学会第 37 回大会, 2020.9.8-10.

参考文献

[1] Brenda Baker. A program for identifying duplicated code. *Proceedings of Computing Science and Statistics: 24th Symposium on the Interface*, Vol. 24, , 07 1992.

[2] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, Vol. 28, pp. 654– 670, 08 2002.

[3] Chanchal Roy and James Cordy. A survey on software clone detection research. *School of Computing TR 2007-541*, 01 2007.

[4] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: scalable and accurate tree-based detection of code clones. pp. 96–105, 06 2007.

[5] James Cordy and Chanchal Roy. The nicad clone detector. pp. 219–220, 06 2011.

[6] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal Roy, and Cristina Lopes. Sourcerercc: scaling code clone detection to big-code. pp. 1157–1168, 05 2016.

[7] Nicholas Kraft, Brandon Bonds, and Randy Smith. Cross-language clone detection. pp. 54–59, 01 2008.

[8] Farouq Al-omari, Iman Keivanloo, Chanchal Roy, and Juergen Rilling. Detecting clones across microsoft .net programming languages. pp. 405–414, 10 2012.

[9] Xiao CHENG, Zhiming PENG, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. Clcminer: Detecting cross-language clones without intermediates. *IEICE Transactions on Information and Systems*, Vol. E100.D, pp. 273–284, 02 2017.

[10] Tijana Vislavski, Gordana Rakić, Nicolás Cardozo, and Zoran Budimac. Licca: A tool for cross-language clone detection. 03 2018.

[11] Daniel Perez and Shigeru Chiba. Cross-language clone detection by learning over abstract syntax trees. pp. 518–528, 05 2019.

[12] Kawser Nafi, Tonny Kar, Banani Roy, Chanchal Roy, and Kevin Schneider. Clcdsa: Cross language code clone detection using syntactical features and api documentation. pp. 1026–1037, 11 2019.

[13] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, Vol. 9, pp. 1735–80, 12 1997.

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan

Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 06 2017.

[15] Zoran Budimac, Gordana Rakić, and Milovs Savi'c. Ssqsa architecture. pp. 287–290, 09 2012.

[16] Zellig Harris. Distributional structure. *Word*, Vol. 10, pp. 146–162, 08 1954.

[17] J Firth. A synopsis of linguistic theory 1930-55. *Studies in linguistic analysis. The Philological Society, Oxford*, pp. 1–32, 01 1957.

[18] Warren Weaver. Translation. *Machine Translation of Languages. Cambridge, Massachusetts: MIT Press*, p. 15–23, 01 1955.

[19] Tomas Mikolov, Ilya Sutskever, Kai Chen, G.s Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *Advances in Neural Information Processing Systems*, Vol. 26, , 10 2013.

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 10 2018.

[21] G.E. Hinton and R.R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science (New York, N.Y.)*, Vol. 313, pp. 504–7, 08 2006.

[22] Philip Gage. A new algorithm for data compression. *The C Users Journal*, Vol. 12, pp. 23–38, 02 1994.

謝辞

本研究を進めるにあたり，毎週ミーティングを行い，研究方針や論文構成，発表手法および練習など多岐にわたって助言と指導をしていただいた指導教員の千葉滋教授に心より感謝致します．また，輪講の練習として，skip-gram でプログラミング言語の分布仮説を検証するという課題を与えてくださった松永智將氏に感謝致します．その課題のお陰で，教師なし学習に興味を惹かれて，本研究のきっかけとなった．そして，研究生活にわたり，たくさんの議論や支援をしてくださった山崎徹郎氏，池崎翔哉氏，李森曦氏，姚旭楊氏，戴峰氏に感謝致します．最後に，日頃から研究室生活やミーティングを通じてご支援くださった千葉研究室の皆様に心より感謝致します．