修士学位論文

題目

自然さを用いたソースコードの定量的評価

指導教員

楠本 真二 教授

報告者

有馬 諒

平成 31 年 2 月 6 日

大阪大学 大学院情報科学研究科

コンピュータサイエンス専攻

平成 30 年度 修士学位論文

自然さを用いたソースコードの定量的評価

有馬 諒

内容梗概

ソースコード解析の研究において，“ソースコードは自然言語と同様の特徴を持つ．そのため自然言語処理手法はソースコードに対しても有用である” という，自然さ仮説が提唱されている．この仮説をもとに，ソースコードに対して自然言語処理手法を適用した研究が近年数多く行われている．その中でも言語モデルを用いたソースコードの自然さ解析は，コード補完やバグ検出で成果が報告されている．自然さとはソースコードがどの程度自然かを数値で表した指標であり，言語モデルを用いて計算される．多くのプロジェクトに存在するソースコードや出現頻度の高いソースコードほど，自然さが高くなる傾向がある．本研究では自然さを用いてソースコードを定量的に評価する手法を提案する．

また，本研究ではリファクタリング支援に対する提案手法の活用を検討した．まず，リファクタリングによってソースコードの自然さがどのように変化しているか，オープンソースソフトウェアである JUnit のソースコードに対して実際に行われたリファクタリングを対象に調査を行った．その結果，28 個中 19 個のリファクタリングにおいて自然さが向上していたことを確認した．次に，自然さがリファクタリングすべき箇所の抽出に有用かを検証した．その結果，自然さを用いることで用いない場合に比べて 40% 効率的に抽出することができ，自然さがリファクタリングすべき箇所の抽出に有用であることを示した．

主な用語

自然さ，言語モデル，リファクタリング

目次

1 はじめに 1

2 準備 4

2.1 自然さ 4

2.2 言語モデル 4

2.3 リファクタリング支援 6

3 自然さ計測手法 7

3.1 トークン切り出し 7

3.2 トークンごとの生成確率計算 8

3.3 トークンごとの値の集約 9

3.4 適用例 11

4 リファクタリングへの適用 14

4.1 リサーチクエスチョン 15

4.2 データセットの構築 15

4.3 リサーチクエスチョンの評価 18

4.4 まとめ 25

5 妥当性の脅威 27

5.1 学習用ソースコードの品質 27

5.2 リファクタリングに関する実験 27

6 おわりに 28

謝辞 30

参考文献 31

図目次

1 自然さのイメージ 4

2 提案手法の概要 7

3 字句解析の実行例 8

4 生成確率の対数を用いる方法 9

5 交差エントロピーを用いる方法 10

6 自然さ計算の例 11

7 リファクタリング対象ソースコード 12

8 メソッド切り出し後のソースコード 12

9 一時変数の導入後のソースコード 13

10 対立するリファクタリングの例 15

11 行データセットの構築方法 17

12 リファクタリングによる自然さの変化量 19

13 自然になったリファクタリング 20

14 自然にならなかったリファクタリング 22

15 リファクタリングによって削除される行の自然さ 24

16 リファクタリングによって削除される行の自然さ 25

表目次

1	JUnit リポジトリの詳細	16
2	対象プロジェクトの概要	18
3	データセットの概要	19
4	すべてのコミットでの結果．✓ は良いリファクタリングと評価されたコミットである．	21

1 はじめに

近年，GNU プロジェクト[*1]や Apache[*2]などのようなオープンソースソフトウェアの普及により，誰もが巨大なソースコードを手に入れることが可能になっている．さらに，GitHub[*3]のようなソースコードホスティングサービスの出現により，10,000 プロジェクト，300,000,000 LOC，1,500,000,000 トークンからなるソースコードデータセットも構築可能となった [1]．また多くのオープンソースソフトウェアでは，ソースコードだけでなくバージョン管理システムによるソフトウェアの開発履歴やバグ管理システムによるバグ情報など，ソフトウェア開発におけるメタ情報も利用することができる．これらのデータセットは，ビッグコードと呼ばれている [2].

ビッグコードのような巨大なソースコードデータセットを解析するにあたって，Hindle らはソースコードの自然さに関する以下の仮説を提唱している [3].

> ソースコードも自然言語と同様に人によって読み書きされる．ゆえにソースコードは自然言語と同様の特徴を持ち，統計的自然言語処理における手法はソースコードに対しても有用である．

統計的自然言語処理とは,「自然言語は短いスニペットの繰り返しであり，そのため出現パターンを統計的手法によって予測可能である」という仮説に基づき，コーパスと呼ばれる大規模な自然言語のデータセットに対して統計的手法や機械学習を用いる手法である [4]．統計的自然言語処理は，機械翻訳や音声認識などの分野で成果が得られている．

Hindle らは統計的自然言語処理手法の 1 つである言語モデルをソースコードに適用した．その結果，ソースコードの言語モデルは英語の言語モデルよりもエントロピーによる指標で優れていることがわかった．Hindle らの提唱した仮説は**自然さ仮説**と呼ばれ [2]，この研究成果をもとに，コード補完 [3, 5, 6] や識別子名の推薦 [7]，構文エラーの修正 [8] などで成果が報告されている．

自然さ仮説に基づく応用研究の 1 つとして，自然さの観点からソースコードを評価する手法がある．Allamanis らは言語モデルを用いて計算される自然さを，ソースコード複雑性メトリクスとして用いることを提案した [1]．Campbell らは，構文エラーを含むソースコードは自然さが低いことを示し，構文エラー箇所の推定に自然さを用いる手法を提案した [9]．Ray らはバグを含むソースコードは自然さが低いことを示し，自然さを用いたバグ検出支援手法を提案した [10]．Hellendoorn らは GitHub におけるソフトウェア開発において，自然さが低い Pull Request はプロジェクトに取り込まれにくいことを示した [11].

[*1] https://www.gnu.org/
[*2] https://www.apache.org/
[*3] https://github.com/

これらの研究成果から，ソースコードの自然さと品質には関連があり，自然さの観点からソースコードを評価することは有効であると考えた．そこで本研究では，自然さを用いたソースコードの定量的評価手法について検討する．ソースコードを評価する手法としてはソフトウェアメトリクス [12, 13] やソースコードの不吉なにおい [14] がある．これらの手法に比べて，自然さを用いる手法では学習用のソースコードをもとに特徴を自動で判別するため，プロジェクトや開発者に応じた評価が可能であると考えられる．

本研究では，自然さによるソースコード評価手法の新たな応用として，自然さをリファクタリング支援に適用する．リファクタリングとはソフトウェアの外部的振る舞いを保ちながら内部構造を改善する作業であり，ソフトウェアの保守性を向上させるために重要な作業である．しかし，ソースコード中のどの部分にどのようなリファクタリングを適用するかは，開発者の経験や勘に頼る場合が多く，リファクタリングを行うことを難しくしている．そのため，ソフトウェアメトリクスやソースコードの不吉なにおいをもとにリファクタリングを支援する手法が提案されている [15]．しかし，実際のソフトウェア開発で行われるリファクタリングでは必ずしもソースコードの不吉なにおいは改善しないことがわかっている [16]．

本研究では自然さをリファクタリング支援に用いることができるかを評価するために実験を行った．まず，リファクタリングとソースコードの自然さに関連があるかを調査するために，実際に行われたリファクタリングにおいて，その前後で自然さがどう変化しているかを調査した．調査対象は JUnit4 に対して実際に行われた 28 個のリファクタリングとした．調査の過程で，[1] や [10] で用いられているエントロピーに基づく自然さでは，うまく評価できない場合があることがわかった．そこで本研究では，一旦ソースコードの行ごとに自然さを計測し，その結果をもとに全体の自然さを求める手法を提案した．提案した手法を用いて，調査対象のリファクタリングにおける自然さの変化を計測した結果，28 個のリファクタリングのうち 19 個のリファクタリングにおいて自然さが向上しており，リファクタリングを行うことによって自然さが向上する傾向があることがわかった．

次に，細かい粒度でリファクタリングと自然さの関連を調査するため，行レベルのデータセットを構築した．このデータセットでは，開発履歴のあるコミットの時点でのスナップショットに対して，どの行がリファクタリングによって削除されるかを識別した．構築には，Ray らのバグ検出の実験 [10] を参考に，SZZ アルゴリズム [17] を元にした手法を用いた．構築したデータセットを用いて，リファクタリングによって削除される行の自然さを，それ以外の行と比較した．その結果，リファクタリングによって削除される行は，そうでない行に比べて，中央値で 0.81 エントロピーが高く，自然さが低い傾向があることがわかった．

さらに，自然さを用いてリファクタリングすべき行を順位付けできるかどうかを調査した．その結果，AUCEC による評価指標が自然さを用いない場合に比べて，AUCEC による評価指標で 73% 高い性能

を得ることができ．自然さはリファクタリングすべき箇所の絞り込みに有用であることがわかった．

本論文では，2 章で準備として言語モデルを用いた自然さについて述べる．また，既存のリファクタリング支援手法について述べる．3 章で自然さを用いたソースコードの評価手法を提案する．4 章で提案手法を用いたリファクタリング支援手法について述べる．5 章で妥当性の脅威について述べ，最後に 6 では本研究のまとめと今後の課題について述べる．

2 準備

2.1 自然さ

本研究における**自然さ**とは，与えられた文がコーパスに対してどれだけ自然かを表した値である．コーパス中に存在しそうな文であるほど自然さは高く，存在しなさそうな文であるほど自然さは低くなる．

例を図 1 に示す．この例において，「A cat caught a」の次に何の単語が続くと自然な文となるかを考える．次の単語が「mouse」であれば，文法的にも意味的にも正しい文となるので，他の単語が続く場合に比べて自然さは高くなる．しかし，次の単語が「mice」の場合，「a」のあとに複数形の単語が続くのは文法的に誤りであるため，自然さは低くなる．また，「mouth」の場合，文法的には正しい文にはなるが，意味がおかしな文となってしまうため，この場合も自然さは低くなる．

自然言語処理ではこのような特徴を文の評価に用いている．例えば機械翻訳では，翻訳の過程で生じた複数の翻訳候補の順位付けに自然さを用いている．

2.2 言語モデル

自然さを数値によって定量的に表すために言語モデルが用いられる．言語モデルとは文に対してその生成確率を割り当てる確率モデルである．文 $S = w_1 w_2 \cdots w_m$ が与えられたときその生成確率は，各単語の条件付確率の積を用いて次のように表される．

$$P(S) = P(w_1) \prod_{i=2}^{m} P(w_i | w_1, \cdots, w_{i-1})$$

ここで $P(w_i | w_1, \cdots, w_{i-1})$ は，単語列 $w_1, \cdots, w_{i-1}$ の次の単語が w_i である確率である．

$P(w_i | w_1, \cdots, w_{i-1})$ は，単語列 $w_1, \cdots, w_{i-1}$ の組み合わせが膨大になるため，直接求めることは現実的ではない．そこで，以下のように直前の $n-1$ 個の単語から次の単語の確率によって近似する手法

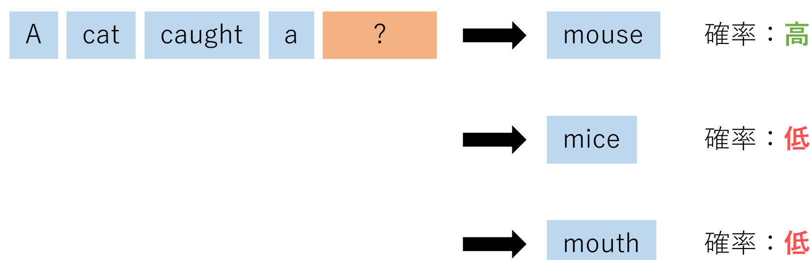


図 1　自然さのイメージ

がある．

$$P(w_i|w_1,\cdots,t_{w-1}) \simeq P(w_i|w_{n-i+1},\cdots,w_{i-1})$$

$P(w_i|w_{n-i+1},\cdots,w_{i-1})$ はコーパス中の単語列の頻度から，最尤推定によって以下のように求められる．

$$P(w_i|w_{n-i+1},\cdots,w_{i-1}) = \frac{C(w_{n-i+1},\cdots,w_{i-1},w_i)}{C(w_{n-i+1},\cdots,w_{i-1})}$$

ここで $C(w_{n-i+1},\cdots,w_{i-1})$ はコーパス中で単語列 $w_{n-i+1},\cdots,w_{i-1}$ が出現した回数を表す．この近似を用いると文の確率 $P(S)$ は以下のようになる．

$$P(S) = \prod_{i=2}^{m} P(w_i|w_{n-i+1},\cdots,w_{i-1}) \tag{1}$$

このように，直前の $n-1$ 個の単語のみを考慮する言語モデルは *n-gram* 言語モデルと呼ばれる．

n-gram 言語モデルの定義は式 (1) であるため，コーパス中に一度も出現しない単語列の確率は 0 となってしまう．これは性能に悪影響を与えるため，出現頻度が 0 の単語に小さい確率を与えるスムージングという手法が用いられる．スムージングには様々な手法が存在する [18]．Hindle らはソースコードに対しては Modified Kneser-Ney スムージングが有効であると述べており [3]，同様の結論が Jimenez らによっても得られている [19]．一方 Tu ら [5] や Ray ら [10] は Katz スムージングを用いている．Hellendoorn らは複数のスムージング手法に対して比較を行い，ソースコードに対しては Jelinek-Mercer スムージングが良いという結論を示している [20]．

n-gram 言語モデルの改良として，Tu らはキャッシュを用いた n-gram 言語モデルを提案した [5]．ソースコードには，ローカル変数のようにソースコードの一部分のみで局所的に出現する単語が存在する．この特性を反映するため，通常の n-gram 言語モデルとキャッシュと呼ばれる計測対象の単語の近傍のみを考慮した言語モデルを以下の式のように組み合わせる手法を提案した．

$$P(w_i|h,cache) = \lambda P_{n\text{-}gram}(w_i|h) + (1-\lambda)P_{cache}(w_i|h)$$

ここで $h = w_{i-n+1},\cdots,w_{i-1}$ であり，w_i の前の単語列である．また，$P_{n\text{-}gram}(w_i|h)$ は通常の n-gram 言語モデル，$P_{cache}(w_i|h)$ はキャッシュにおける w_i の生成確率である．

Hellendoorn らはキャッシュを用いた n-gram 言語モデルをさらに発展させ，複数のモデルを入れ子に構成した言語モデルを提案した [20]．

n-gram 言語モデルにおいて，n を大きくすると指数的に学習コストが増加する．そのため，n を大きくすることは難しく狭い範囲の文脈のみしか学習できないという問題点がある．この問題を解決する手法として自然言語処理分野では深層学習を用いた言語モデルが提案されている [21, 22]．中でも LSTM[23] と呼ばれる構造を用いたニューラルネットワークによる言語モデルでは，状態をネットワーク内に保持することによってより長い文脈を考慮することが可能になった [24]．

White らは深層学習を用いた言語モデルをソースコードに適用した [25]．これらの深層学習を用いたモデルをソースコードに合わせて改良したモデルも提案されている [26, 27]．

深層学習による言語モデルを用いた応用が盛んに研究されており，エラー修正 [28, 29, 30]，ソースコード要約 [31]，コミットメッセージ生成 [32]，メソッド名生成 [33] で成果が報告されており，Allamanis らはこれらの研究をまとめたサーベイ論文を発表している [2]．

2.3 リファクタリング支援

近年，ソフトウェアの大規模化，複雑化によってソフトウェア保守の作業量は増加しており，ソフトウェアの保守性を保つためにもソースコードの品質は重要である．しかし，機能の追加や変更，バグ修正などによって，ソースコードには変更を加えられ続けるため，ソースコードの品質は徐々に低下していく [34]．このような品質の低下を抑えるために，リファクタリング [14] という作業が行われる．リファクタリングとはソフトウェアの外部的振る舞いを保ちながら内部構造を改善する作業であり，ソフトウェアの保守性を向上させるために重要な作業である．ソースコードの不吉な臭いとは，将来問題を引き起こす可能性のある箇所を指し，リファクタリングすべき場所であると言われている．

リファクタリングすべき不吉な臭いの検出に関する研究が行われている．手法として，コード行数やサイクロマチック数 [12]，CK メトリクス [13] などのソフトウェアメトリクスを用いる手法が存在する [15]．Palomba らはこれらのメトリクスに加えて開発履歴のメトリクスを用いて不吉な臭いを検出する方法を提案している [35]．

しかし，ソフトウェア開発で実際に行われるリファクタリングでは，メトリクスに基づくソースコードの不吉な臭いは改善しないことがわかっている [16, 36, 37]．Silva ら [38] は，リファクタリングを行った理由について開発者へインタビューを行い，不吉な臭いに基づくリファクタリングが少ないことを明らかにした．

他のアプローチとして，テキスト解析による手法 [39]，機械学習による手法 [40, 41] が提案されている．

3　自然さ計測手法

本章では，本研究で用いたソースコードの自然さを計測する手法について述べる．手法の入力は計測対象のソースコードであり，出力は自然さである．

提案手法では以下の手順で自然さを計測する．

1. トークン切り出し
2. トークンごとの生成確率計算
3. トークンごとの値の集約

提案手法の概要を図 2 に示す．

3.1　トークン切り出し

提案手法ではまず，入力のソースコードに対して字句解析を行い，ソースコードをトークン列に変換する．この際にコメントや空白などは取り除かれ，トークンには含まれない．字句解析の際に問題となるのは，識別子名の扱いである．ソースコード中のクラス名やメソッド名，変数名などの識別子名は開発者が自由に決めることができ，多くの場合複数の英単語を組み合わせて 1 つの識別子名とする．その

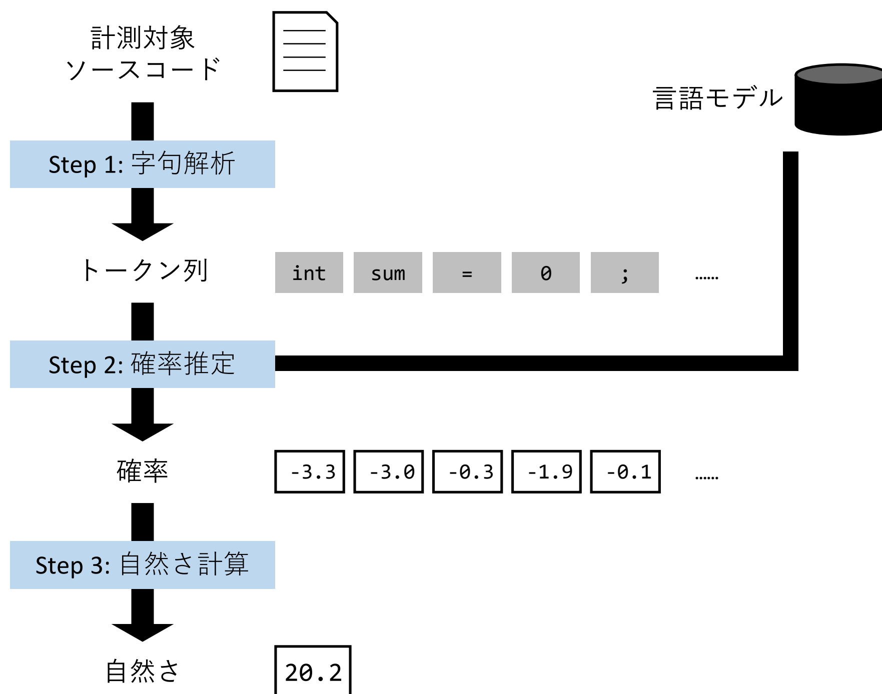


図 2　提案手法の概要

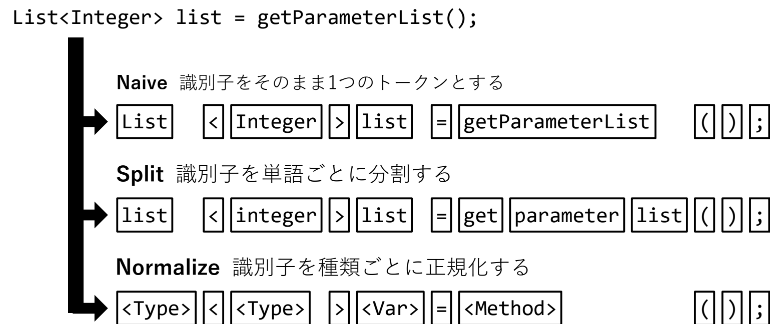


図 3　字句解析の実行例

ため，ソースコード中に含まれる識別子名の種類が膨大になり，うまくモデルを推定できない可能性がある [20]．また，似た単語から構成される識別子も全く別の識別子としてみなしてしまうという問題もある．そこで，以下の 3 つの戦略で識別子名を扱う．

Naive　識別子をそのまま 1 つのトークンとする．

Split　識別子を単語ごとに分割する

Normalize　識別子を種類ごとに正規化する．

3 つの戦略による字句解析の例を図 3 に示す．

Naive 戦略では，字句解析によって得られた識別子をそのまま単語として用いる．

Split 戦略では，`getParameterList` のように複数単語からなる識別子名を `get`，`parameter`，`list` のように単語ごとに分割する．これによって，似た目的の識別子を言語モデルがうまく理解できるようになると考えられる．

Normalize 戦略では `Integer` は`<Type>`，`getParameterList` は`<Method>`のように，その識別子の種類ごとに同じトークンとして正規化される．これによって，ソースコードの構造的な特徴を言語モデルによって学習できると考えられる．

3.2　トークンごとの生成確率計算

次に得られたトークン列の各トークンの生成確率を言語モデルを用いて計算する．

言語モデルには Tu ら [5] によって提案されたキャッシュ付き n-gram 言語モデルを用いる．パラメータとしては，$n = 10$，キャッシュの対象は同一ファイル内のすべて，スムージング手法は Hellendoorn らによって良い結果が報告されている Jelinek-Mercer スムージングを用いる．また，Ray らのバグ検出における実験 [10] において，トークン列を前から順に走査したときの生成確率だけでなく，逆順に走査したときの生成確率を組み合わせること性能が向上したと報告されているため本研究でも採用した．

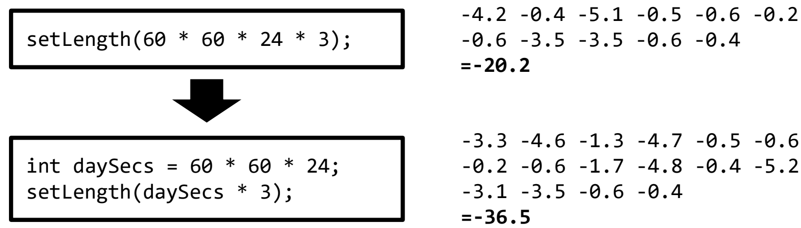


図 4　生成確率の対数を用いる方法

言語モデルの構築に用いるコーパスには，対象プロジェクトに含まれるソースコードを交差検証によって用いる方法と，対象プロジェクトとは別のプロジェクト群のソースコードを用いる方法がある．

最後に，構築した言語モデルを用いてトークン列のトークンごとの生成確率を計算する．

3.3　トークンごとの値の集約

最後に，トークンごとの生成確率を，行，メソッド，ファイルなど必要な粒度に応じて集約し，自然さを求める．本研究では集約の手法として 3 つの方法を用いた．

対数生成確率　各単語の生成確率の対数を用いる

交差エントロピー　各単語のエントロピーを用いる

提案手法　対数生成確率を閾値で正規化する

対数生成確率

1 つ目はスニペットの生成確率の対数を用いる方法である．文の生成確率 $P(S)$ は言語モデルの定義から以下のとおりである．

$$P(S) = P(w_1)\prod_{i=2}^{m} P(w_i|w_1, \cdots, w_{i-1})$$

$P(w_i|w_1, \cdots, w_{i-1})$ は 10^{-10} のような小さな値となることが多く，それらの積をコンピュータで計算するとアンダーフローを引き起こし，正確な値を計算できない可能性がある．自然さは他の値との大小関係が重要であり絶対値の大きさ自体に意味はないため，以下の式のように確率の対数を取った値を用いることがよく行われる．

$$\begin{aligned}\log P(S) &= \log\left(P(w_1)\prod_{i=2}^{m} P(w_i|w_1, \cdots, w_{i-1})\right)\\ &= \log P(w_1) + \sum_{i=2}^{m}\log P(w_i|w_{i-n+1}, \cdots, w_{i-1})\end{aligned}$$

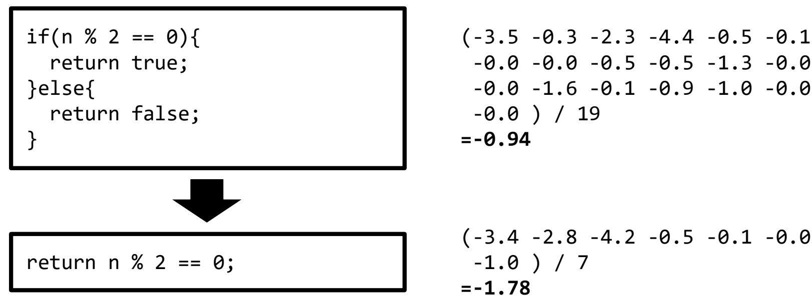


図 5　交差エントロピーを用いる方法

この $\log P(S)$ をスニペットの自然さとする．対数生成確率による自然さは，機械翻訳等の自然言語処理で多く用いられている．

問題点はトークン列の長さが短いほど自然さが高くなるという点である．その例を，図 4 に示す．(a) のスニペットの自然さは -20.2 であるが，これに対して，一時変数を導入するリファクタリングを行った (b) の自然さは -36.5 となる．この方法を用いるとソースコードは短ければ短いほうが良いことになってしまう．

交差エントロピー

2 つ目はスニペットの交差エントロピーを用いる方法である．交差エントロピーとは単語ごとの情報量の期待値であり，言語モデルがどれだけよく表すことができているかの指標である．交差エントロピー $H(S)$ は以下の式で求められる．

$$H(S) = \frac{1}{n} \log P(S)$$

交差エントロピーはバグ検出 [10] 等で用いられている．

問題点は，予約語や記号が多くなるほど自然さが高くなるという点である．その例を，図 5 に示す．(a) のスニペットの自然さは -0.94 であるが，これに対して，3 項演算子を用いて同様の処理を記述した (b) の自然さは -1.78 となる．“{” や “`true`” のように，ソースコード中によく出現する記号や予約語の生成確率は高いため，多く含まれるほど平均も高くなる．そのため，この方法を用いるとソースコードは冗長に書いたほうが良いということになってしまい，コードクローンの研究と矛盾する．

提案手法

そこで本研究では，3 つ目の手法として行ごとの対数生成確率もとに，以下のように計算する．

$$naturalness = \frac{1}{M} \sum_{i=0}^{M} max(0, threshold - N(L_i))$$

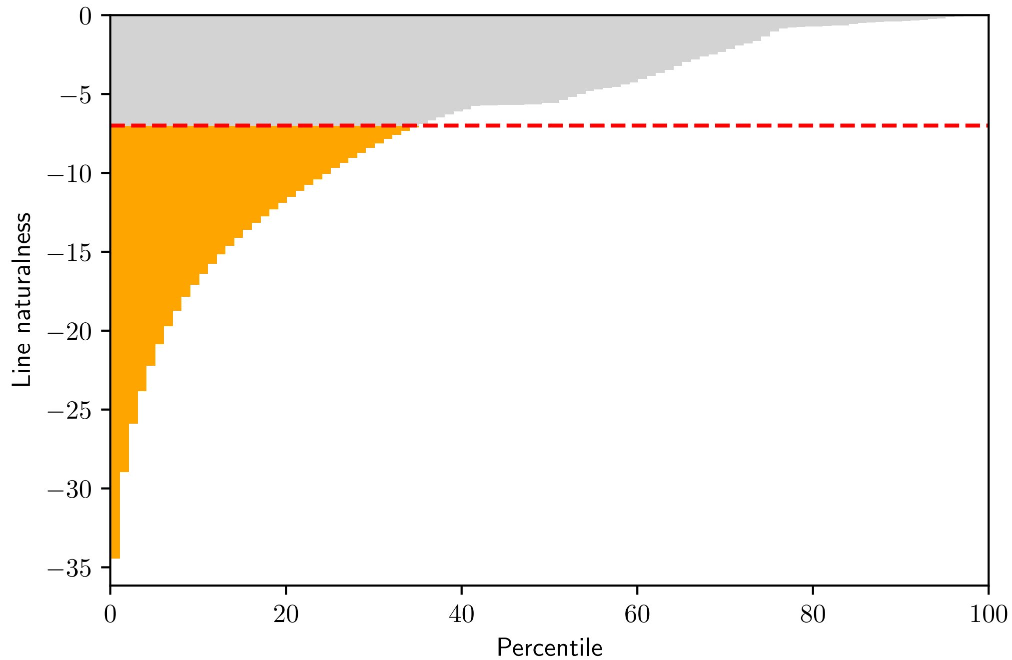


図 6　自然さ計算の例

ここで，M は計測対象ソースコードの行数，$threshold$ は閾値，$N(L_i)$ は i 番目の行の生成確率の対数を表す．本研究では，閾値に学習用ソースコードコーパスにおける自然さの分布の中央値を用いる．この自然さ計算方法では，値が小さいほど自然なソースコードである．

提案手法による自然さの意味を図 6 を用いて説明する．図 6 は，各行の自然さを低い順に並び替え，x 軸 0 から 100 の間にプロットしたグラフである．中央値を破線としたとき提案手法の自然さはこの破線より下の部分の面積であり，面積が小さいほど自然なソースコードとなる．

3.4　適用例

リファクタリングを行った時，提案手法による自然さがどのよう変化するかの例を図 7 のソースコードを用いて示す．このソースコードは 2 つの文字列を 2 桁の 16 進数とみなして加算を行い，その結果を出力するプログラムである．言語モデル生成に用いたソースコードコーパスにおける行ごとの自然さの中央値が −5.7 であったため，このソースコードの自然さは 145.6 であった．

まず，16 進数文字列を数値に変換する処理が重複しているため，この処理をメソッドとして切り出すリファクタリングを行った．この結果を図 8 に示す．このソースコードの自然さは 66.6 であり，リファクタリングによって元のソースコードよりも自然さが増加した．

```
public void add(String left, String right){
  int leftValue = 0;
  for(int i = 0; i < left.length(); i++){
    leftValue *= 16;
    leftValue += 'A'<=left.charAt(i) && left.charAt(i)<=
     'F' ? left.charAt(i)-'A' + 10 : left.charAt(i) - '0';
  }

  int rightValue = 0;
  for(int i = 0; i < right.length(); i++){
    rightValue *= 16;
    rightValue += 'A'<= right.charAt(i) && right.charAt(i) <=
     'F‘ ? right.charAt(i) - 'A' + 10: right.charAt(i) - '0';
  }

  System.out.println(leftValue + " " +  rightValue);
}
```

Line	自然さ
`int leftValue = 0;`	-4.4
`for(int i = 0; i < left.length(); i++){`	-8.7
`leftValue *= 16;`	-6.6
`leftValue += 'A'<=left.charAt(i) && left.charAt(i)<= 'F' ? left.charAt(i)-'A' + 10 : left.charAt(i) - '0';`	-73.6
`}`	-0.8
`int rightValue = 0;`	-3.6
`for(int i = 0; i < right.length(); i++){`	-8.4
`rightValue *= 16;`	-6.6
`rightValue += 'A'<= right.charAt(i) && right.charAt(i) <= 'F‘ ? right.charAt(i) - 'A' + 10: right.charAt(i) - '0';`	-73.8
`}`	-0.8
`System.out.println(leftValue + " " +  rightValue);`	-7.9

図 7　リファクタリング対象ソースコード

```
public void add(String left, String right){
  int leftValue = convertHex(left);
  int rightValue = convertHex(right);
  System.out.println(leftValue + " " +  rightValue);
}

public int convertHex(String str){
  int value = 0;
  for(int i = 0; i < str.length(); i++){
    value *= 16;
    value += 'A' <= str.charAt(i) && str.charAt(i) <= 'F' ?
      str.charAt(i) - 'A' + 10: str.charAt(i) - '0';
  }
  return value;
}
```

Line	自然さ
`int leftValue = convertHex(left);`	-9.3
`int rightValue = convertHex(right);`	-7.8
`System.out.println(leftValue + " " +  rightValue);`	-8.1
`int value = 0;`	-5.3
`for(int i = 0; i < str.length(); i++){`	-5.8
`value *= 16;`	-9.1
`value += 'A' <= str.charAt(i) && str.charAt(i) <= 'F' ? str.charAt(i) - 'A' + 10: str.charAt(i) - '0';`	-60.6
`}`	-0.8
`return value;`	-3.5

図 8　メソッド切り出し後のソースコード

さらに，16 進数の各桁を処理する行は，1 つの行で多くの処理を行っているため，この行を分解するリファクタリングを行った．この結果を図 9 に示す．このソースコードの自然さは 27.3 であり，図 8 のソースコードよりもさらに自然なソースコードとなったといえる．

```java
public void add(String left, String right){
  int leftValue = convertHex(left);
  int rightValue = convertHex(right);
  System.out.println(leftValue + " " +  rightValue);
}

public int convertHex(String str){
  int value = 0;
  for(int i = 0; i < str.length(); i++){
    value *= 16;
    char c = str.charAt(i);
    if('A' <= c && c <= 'F'){
      value += c - 'A' + 10;
    }else{
      value += c - '0';
    }
  }
  return value;
}
```

コード行	自然さ
`int leftValue = convertHex(left);`	-9.3
`int rightValue = convertHex(right);`	-7.8
`System.out.println(leftValue + " " +  rightValue);`	-8.1
`int value = 0;`	-5.3
`for(int i = 0; i < str.length(); i++){`	-5.8
`value *= 16;`	-9.1
`char c = str.charAt(i);`	-6.8
`if('A' <= c && c <= 'F'){`	-7.5
`value += c - 'A' + 10;`	-12.3
`}else{`	-1.5
`value += c - '0';`	-11.8
`}`	-0.8
`}`	-0.8
`return value;`	-3.2

図 9　一時変数の導入後のソースコード

4 リファクタリングへの適用

近年，ソフトウェアの大規模化，複雑化によってソフトウェア保守の作業量は増加しており，ソフトウェアの保守性を保つためにもソースコードの品質は重要である．しかし，機能の追加や変更，バグ修正などによって，ソースコードには変更を加えられ続けるため，ソースコードの品質は徐々に低下していく [34]．このような品質の低下を抑えるために，リファクタリングという作業が行われる．リファクタリングとはソフトウェアの外部的振る舞いを保ちながら内部構造を改善する作業であり，ソフトウェアの保守性を向上させるために重要な作業である．しかし，ソースコード中のどの部分にどのようなリファクタリングを適用するかは，開発者の経験や勘に頼る場合が多く，リファクタリングを行うことを難しくしている．

開発者の経験や勘が必要なリファクタリングの例を図 10 を用いて説明する．図において，左側のソースコードに対して，for 文の内側の処理を別のメソッドとして切り出すリファクタリングを考える．このように，ソースコード中の処理の一部を別のメソッドとして切り出すリファクタリングは Extract Method リファクタリングと呼ばれる．このリファクタリングによって，重複した処理を再利用できること，意味のある処理の単位を 1 つのメソッドとしてまとめて名前をつけることでソースコードの可読性が向上することなどの利点があげられる．

しかし，逆に必要以上に細かく分割されたメソッドは可読性が低下するといわれている．そのため，複数のメソッドに分解された処理を 1 つのメソッドに統合するリファクタリングも行われており，このようなリファクタリングは Inline Method リファクタリングと呼ばれる．

このように Extract Method リファクタリングと Inline Method リファクタリングは互いに対立したリファクタリングであり，どちらを適用するかはメソッドの規模や処理内容などから開発者が判断しなければならない．このような状況において，リファクタリングの評価を何らかの方法で数値化しどちらの状態がよりよいか比較できるようになれば，リファクタリングを行う開発者の支援になると著者らは考える．

そのため，リファクタリングを支援する手法が研究されている．支援の 1 つとして，メトリクスを用いる方法 [15] があるが，実際に行われる多くのリファクタリングにおいてメトリクスは向上しないという結果もあり [16]，リファクタリング支援に対してメトリクスでは不十分であると考えられる．

そこで本研究では，ソースコードをその “自然さ” という観点から評価することによってリファクタリングを支援する手法を検討する．

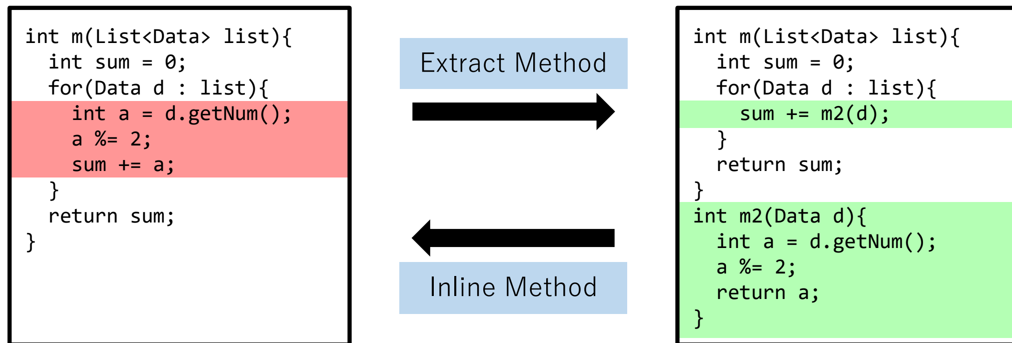


図 10　対立するリファクタリングの例

4.1　リサーチクエスチョン

本研究では，リファクタリングの支援にソースコードの自然さによる評価指標を用いることができるかを評価するために，以下の 3 つのリサーチクエスチョンをたてた．

RQ1: リファクタリングによって自然さはどう変化するか　リファクタリングの前後でソースコードの自然さがどのように変化するか，実際のオープンソースプロジェクトで行われたリファクタリングに対して調査する．自然さがリファクタリングの評価指標として有用ならば，リファクタリングによって自然さが向上していると考えられる．

RQ2: リファクタリングによって削除される行の自然さは低いか　行ごとの比較によって，リファクタリングと自然さの関係を調査する．

RQ3: リファクタリングすべき箇所の絞り込みに自然さは有用か　リファクタリングすべき箇所を絞り込むという観点で，すべての行を自然さの順に並べ替えることで，リファクタリングすべき箇所の発見に役立っているかを評価する

4.2　データセットの構築

リサーチクエスチョンの評価のためのデータセット構築法について説明する．本研究では，2 つのデータセットを構築した．

4.2.1　コミットレベルデータセット

本節では，RQ1 の検証のために構築した，コミットレベルでのリファクタリングデータセットの構築方法について述べる．このデータセットは，リポジトリの中でリファクタリングのみが行われたコミットからなる．

対象プロジェクトとして，JUnit4[*4]を用いた．これは Java 用の単体テスト自動化フレームワークであり，JUnit4 自体も Java によって実装されている．JUnit4 は GitHub によって管理されており，GitHub を用いた開発者相互によるソースコードレビューによってソースコードの品質が担保されている．

データセット構築のため，まずコミットメッセージにリファクタリングに関連があると思われるキーワードを含むコミットを抽出した．本実験ではキーワードとして *refactor*，*clean* を用いた．次に抽出されたコミットを目視で確認し，変更内容がリファクタリングのみであり，処理の内容を変えないコミットのみを抽出した．これによって 28 個のコミットを抽出することができた．対象リポジトリの詳細を表 1 に示す．

4.2.2 行レベルデータセット

本節では，RQ2，RQ3 の検証のために構築した，行レベルでのリファクタリングデータセットの構築方法について述べる．このデータセットでは Ray らのバグに関する実験 [10] を参考に，リポジトリにおける，あるコミット時点のスナップショットにおいて，どの行が将来のリファクタリングによって削除されるかを識別した．

データセットの構築は以下の 3 つの STEP からなる．

STEP1　開発履歴から行われたリファクタリングを抽出
STEP2　リファクタリングの原因となった変更の特定
STEP3　スナップショットコミットにおける行の識別

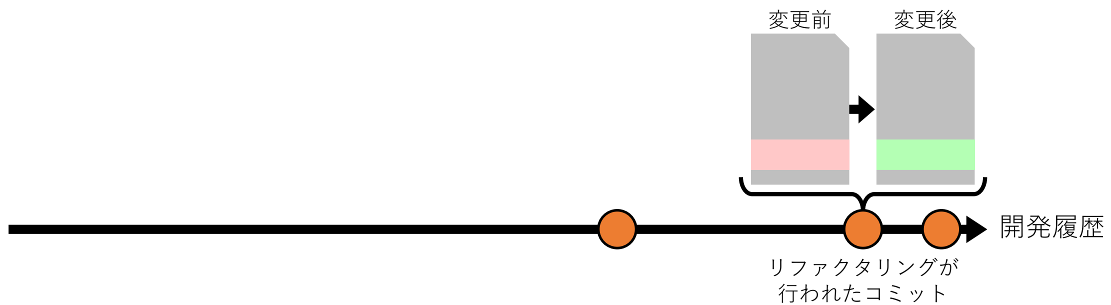
構築手法の概要を図 11 に示す．

表 1　JUnit リポジトリの詳細

期間	2000/12/03 - 2017/10/16
最新コミットでのファイル数	449
最新コミットでの LOC	43161
全コミット数	2195
キーワードを含むコミット数	354
目視確認による リファクタリングコミット数	28

[*4] https://github.com/junit-team/junit4

STEP1：開発履歴から行われたリファクタリングを抽出



STEP2：リファクタリングの原因となった変更の特定

STEP3：スナップショットコミットにおける行の識別

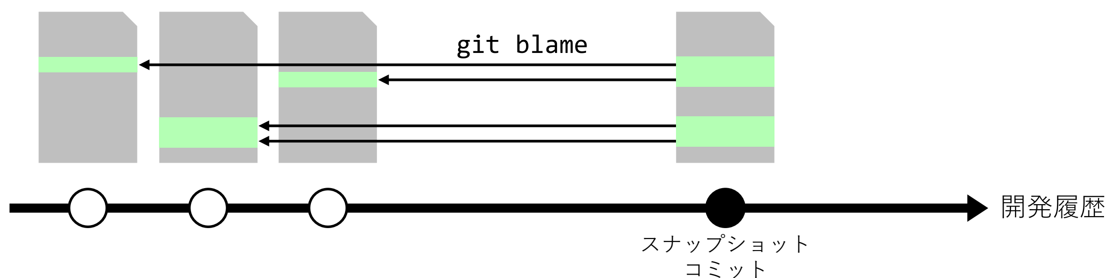


図 11　行データセットの構築方法

対象は，Ray らの実験で用いられた 10 のプロジェクトを用いた．このとき，Qpid は対象プロジェクトのリポジトリが見つからなかったため除外した．また，Elasticsearch，Derby，Lucene は解析中にエラーとなったため除外した．そのため，それ以外の 6 つのプロジェクトを用いた．

STEP1 では，対象のソフトウェアのリポジトリを解析することで，開発中に行われたリファクタリングを抽出する．リファクタリングの抽出には RefactoringMiner[42] を用いた．このツールは，AST

上で差分のマッチングを行うことで，適合率 98%，再現率 87% でリファクタリングを抽出することができる．RefactoringMiner によってリファクタリングを検出した結果，6 つのプロジェクトから合計 5,153 個のリファクタリングを抽出した．

STEP2 では，抽出したリファクタリングに対して，リファクタリングによって削除された行がいつ導入されたかを解析することで，リファクタリングの原因となった変更を特定する．これには SZZ アルゴリズム [17] と呼ばれる，バグの原因コミット特定手法を利用する．この手法では `git-blame` コマンドを用いて，リファクタリングによって削除された行がどのコミットで追加されたかを同定することでリファクタリングの原因となった変更を特定する．

最後に STEP3 では，スナップショットコミットにおけるすべての行で `git-blame` コマンドを実行する．このとき，`blame` コマンドによって検出された行が，STEP2 によって特定されたリファクタリングの原因となった変更であった場合，この行はリファクタリングによって削除される行であると判別できる．この結果，計測対象 9,801,793 行のうち，180,819 行を将来リファクタリングによって削除される行として抽出した．表 2 にデータセットのプロジェクトごとの詳細について示す．

4.3 リサーチクエスチョンの評価

4.3.1 RQ1: リファクタリングによって自然さはどう変化するか

RQ1 では，リファクタリングによって自然さが向上するかを調査する．

RQ1 の評価のために，ソフトウェア開発において実際に行われたリファクタリングに対して，自然さが向上しているかどうかの調査を行った．

自然さ計測に用いる言語モデルは，Higo らによって作成されたデータセット [43] を用いて構築し

表 2　対象プロジェクトの概要

プロジェクト	リファクタリング数	計測対象行数	リファクタリング行数
Atmosphere	417	388,734	10,965
Facebook-android-sdk	188	135,932	2,986
Netty	1,159	909,879	26,761
Presto	1,747	2,477,122	51,338
OpenJPA	713	4,133,333	53,878
Wicket	929	1,756,793	34,891
合計	5,153	9,801,793	180,819

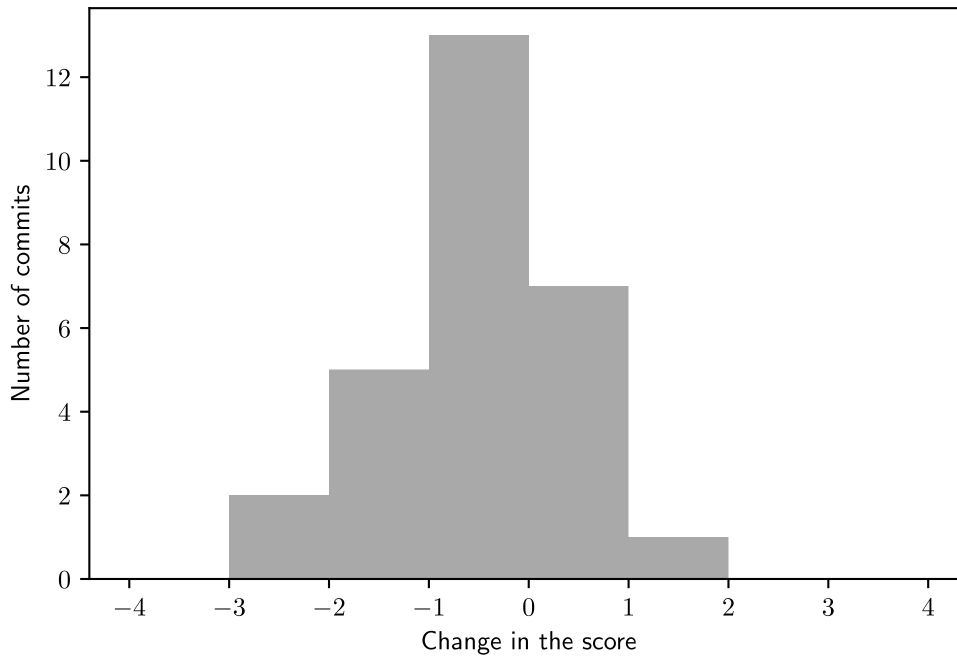


図 12　リファクタリングによる自然さの変化量

た．これは Apache Software Foundation[*5]の Java プロジェクトから作成されたデータセットである．データセットの概要を表 3 に示す．

抽出した各コミットについて，変更前のソースコードと，変更後のソースコードのそれぞれに対して提案手法を適用し，それぞれの自然さを求めた．このとき自然さ計算に必要な閾値は，学習用ソースコードコーパスに対して交差検定を行って行ごとの自然さの分布を求め，その中央値を用いた．こうして求めた自然さが変更の前後で増加しているか，減少しているかを確認した．

実験の結果，28 個のリファクタリングのうち 19 個で提案手法による自然さが向上した．すべてのリ

表 3　データセットの概要

プロジェクト数	84
ファイル数	66,724
LOC	11,545,556

[*5] http://www.apache.org/

```
+ private List getAnnotatedFieldsByParameter() {
+  return getTestClass().getAnnotatedFields(Parameter.class);
+ }

+ private boolean fieldsAreAnnotated() {
+  return !getAnnotatedFieldsByParameter().isEmpty();
+ }
```

自然さ: -50.5（2行目）, -44.2（6行目）

(a) 切り出されたメソッド

```
 @Override
 protected void validateConstructor(List<Throwable> errors) {
    validateOnlyOneConstructor(errors);
-   List<FrameworkField> annotatedFieldsByParameter =
       getTestClass().getAnnotatedFields(Parameter.class);
-   if (annotatedFieldsByParameter.size() > 0) {
+   if (fieldsAreAnnotated()) {
        validateZeroArgConstructor(errors);
    }
  }

  @Override
  protected void validateFields(List<Throwable> errors) {
    super.validateFields(errors);
-   List<FrameworkField> annotatedFieldsByParameter =
       getTestClass().getAnnotatedFields(Parameter.class);
-   if (annotatedFieldsByParameter.size() > 0) {
+   if (fieldsAreAnnotated()) {
+      List<FrameworkField> annotatedFieldsByParameter =
       getAnnotatedFieldsByParameter();
```

自然さ: -19.2, -84.4, -62.3, -32.1, -2.9, -37.1, -90.8, -60.3, -36.7, -56.7

(b) メソッド呼び出しへの変更

図 13　自然になったリファクタリング

ファクタリングでの結果を，表 4 に示す．✓ はリファクタリングによって自然さが向上したコミットである．

この結果から，多くのリファクタリングで開発者によって行われたリファクタリングと，提案手法によって良いと評価されたリファクタリングが対応しているといえる．またこれらのリファクタリングに対して，リファクタリングによる自然さの変化量を計算した．そのヒストグラムを図 12 に示す．このヒストグラムからも，リファクタリングによってソースコードの自然さが高くなる傾向があることがわかる．

リファクタリングによってソースコードの自然さが向上した例として，2012/8/20 15:47 のコミット#0215c66 を図 13 に示す．このコミットでは Extract Method リファクタリングが行われており，図 13(a) に示す 2 つのメソッドが切り出されている．このとき，行の自然さ-50.3 は交差検定による分布中では下位 14% に相当し，-44.2 は下位 18% に相当する．切り出した 2 つのメソッドを使用するように変更された部分を図 13(b) に示す．この変更では，自然さの低かった行がメソッド呼び出しに変更されることで自然さが向上していることがわかる．このように，自然さが低く繰り返し出現する行をメソッ

表 4　すべてのコミットでの結果．✓ は良いリファクタリングと評価されたコミットである．

コミット ID	リファクタリングの内容	自然さの変化	評価
#23793cd	Extract class	0.46	
#a7c4d03	Extract class	-1.76	✓
#7a2b046	Extract class	-0.62	✓
#fd1ef3c	Extract method	-0.21	✓
#dbe7711	Extract method	-0.49	✓
#2240984	Extract method	0.07	
#862f41c	Extract method	0.91	
#5976b1d	Extract method	-0.15	✓
#0215c66	Extract method	-2.78	✓
#24cbcbc	Extract method	-2.19	✓
#467dd07	Extract method	-0.26	✓
#e48f6d4	Extract method	-1.09	✓
#fe5d86e	Inline method	-1.01	✓
#6838ac0	Inline method	0.49	
#0030e51	Inline method	1.02	
#ce9bc58	Move method	0.00	
#f1f4fe2	Move method	-0.72	✓
#4c1758d	アルゴリズム変更	-0.38	✓
#66bfb24	アルゴリズム変更	-1.14	✓
#df016dc	アルゴリズム変更	-0.24	✓
#17a2f11	不要な処理の除去	-1.15	✓
#9a0aec8	不要な継承の除去	0.00	
#a30e87b	変数に final を付与	0.00	
#db8d580	コードクローンの除去	-0.49	✓
#e77e1c4	既存メソッドの再利用	-0.23	✓
#759061a	Strategy パターンの導入	-0.04	✓
#7f2569f	Pull up method	-0.46	
#d064212	Rename method and class	-0.50	✓

```
+ private Collection<T> getFilteredChildrenWithoutIgnores(              自然さ
+             final RunNotifier notifier) {
+   final Collection<T> filteredChildren                               -60.1
      = getFilteredChildren();
+   Collection<T> filteredChildrenCopy                                 -55.3
      = new ArrayList<T>(filteredChildren);
+
+   for (T child : filteredChildren) {                                 -47.6
+     if (isIgnoredMethod(child)) {                                    -43.7
+       Description childDescription= describeChild(child);            -29.2
+       notifier.fireTestIgnored(childDescription);                    -29.6
+       filteredChildrenCopy.remove(child);                            -31.4
+     }                                                                -14.1
+
+   }                                                                   -6.2
+
+   return Collections                                                 -20.6
+           .unmodifiableCollection(filteredChildrenCopy);             -24.5
+
+ }
```

(a) 切り出されたメソッド

```
  protected Statement childrenInvoker(final RunNotifier notifier) {   自然さ
-   final Collection<T> filteredChildren= getFilteredChildren();       -61.6
-  Collection<T> filteredChildrenCopy=                                 -56.0
      new ArrayList<T>(filteredChildren);
-
-   for (T child : filteredChildren) {                                 -48.2
-     if (isIgnoredMethod(child)) {                                    -44.2
-       Description childDescription= describeChild(child);            -29.4
-       notifier.fireTestIgnored(childDescription);                    -29.9
-       filteredChildrenCopy.remove(child);                            -31.8
-     }                                                                -14.3
-
-   }                                                                   -6.2
-
-   final Collection<T> filteredChildrenWithoutIgnores=Collections    -23.7
-          .unmodifiableCollection(filteredChildrenCopy);              -24.7
+   final Collection<T> filteredChildrenWithoutIgnores=                -65.2
      getFilteredChildrenWithoutIgnores(notifier);

    if (filteredChildrenWithoutIgnores.isEmpty()) {                    -53.0
      return new EmptyStatement();                                     -29.0
    }                                                                  -12.8

    return new Statement() {                                           -31.2
      @Override
      public void evaluate() {
        runChildren(notifier, filteredChildrenWithoutIgnores);         -31.2
      }
    };                                                                 -14.4
  }
```

(b) メソッド呼び出しへの変更

図 14　自然にならなかったリファクタリング

ドとしてまとめることで自然さの低い行が削減されたため，全体の自然さが向上したと考えられる．

リファクタリングによってソースコードの自然さが低下した例として，2013/10/21 01:58 のコミット`#2240984` を図 14 に示す．このコミットにおいても Extract Method リファクタリングが行われており，図 14(a) に示す `getFilteredChildrenWithoutIgnores` メソッドが切り出されている．切り出したメソッドを使用するように変更された部分を図 14(b) に示す．この変更では，Extract Method リファクタリングによって新たなメソッドが追加されたものの，ソースコード中に存在する行自体はほぼ変わっておらず，新たなメソッド呼び出しが追加された分だけ自然さが低下した．提案手法は n-gram によって行単位の自然さを計測しているため，各行が自然か，そうでないかは判定できるが，それら行の並びが自然かそうでないかはほぼ考慮することができない．そのため図 14 のようなリファクタリングを評価することは難しい．

RQ1 の結論：リファクタリングによって自然さは向上する傾向がある．

4.3.2 RQ2: リファクタリングによって削除される行の自然さは低いか

6 つのプロジェクトから得られたリファクタリングによって削除される行のデータセットに対して，他の部分に比べて自然さが低いかをエントロピーを用いて検証した．

結果の箱ひげ図を，図 15 に示す．リファクタリングによって削除される行のエントロピーはの中央値は 4.40，それ以外の行のエントロピーの中央値は 3.59 となり，リファクタリングによって削除される行は自然さが低い傾向があることがわかった．

リファクタリングによって削除される行のエントロピーとそれ以外の行のエントロピーの間に有意差があるか，マンホイットニーの U 検定を行った．この結果，$p < 10^{-12}$ となり有意差があることがわかった．これより，リファクタリングによって削除される行はそれ以外の行に比べて自然さが低いと言える．

また，2 つのエントロピー間の効果量として Cohen の d[44] を計算した結果，0.14 となった．この値は，Ray らのバグに関する研究 [10] において，バグを含む行とそうでない行のエントロピーの差について解析した際の効果量 (0.26) と近い値である．そのため効果量の観点からも，リファクタリングによって削除される行はそれ以外の行に比べて自然さが低いということが言える．

RQ2 の結論：リファクタリングによって削除される行の自然さは他と比べて低い．

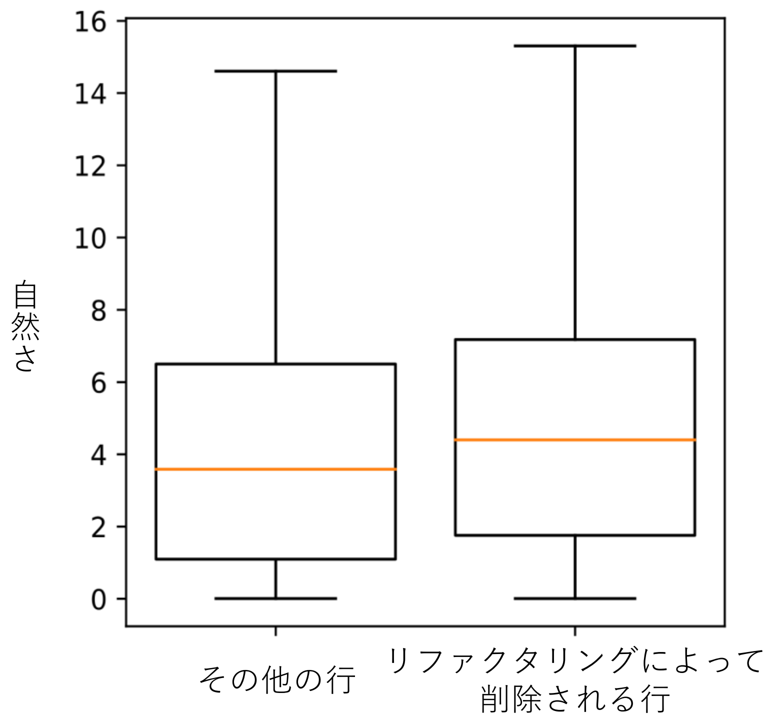


図 15　リファクタリングによって削除される行の自然さ

4.3.3　RQ3: リファクタリングすべき箇所の絞り込みに自然さは有用か

自然さによるリファクタリングすべき箇所の絞り込み性能を評価するために，AUCEC[45] という指標を用いる．これは，どれだけ効率的にリファクタリングすべき箇所を絞り込むことができているかを表す指標である．この指標を計算するためにまず，対象のソースコードを自然さの低い順に並び替える．そして，x 軸を調査済みソースコードの割合，y 軸を見つかったリファクタリングによって削除される行の割合としてグラフ上に曲線を描く．このとき比較のため，並べ替えをランダムに行ったときの期待値として，x=y の直線と比較する．このグラフにおける曲線の下の部分の指標が AUCEC である．リファクタリングを行う際に，すべての行を調査する設定は現実的ではないので，ある割合までの AUCEC を求めることが多い．今回は Ray らと同様に 0.2 を用いた．

図 16 に 6 つのプロジェクトから得られたリファクタリングによって削除される行のデータセットグラフを示す．このグラフから，自然さを用いたときの曲線がランダムよりも上側を通っていることがわかる．0.2 までを用いた AUCEC の値も 0.035 とランダムに比べて 73% 増加した．このことから，リファクタリングすべき箇所の絞り込みに自然さは有用であると言える．

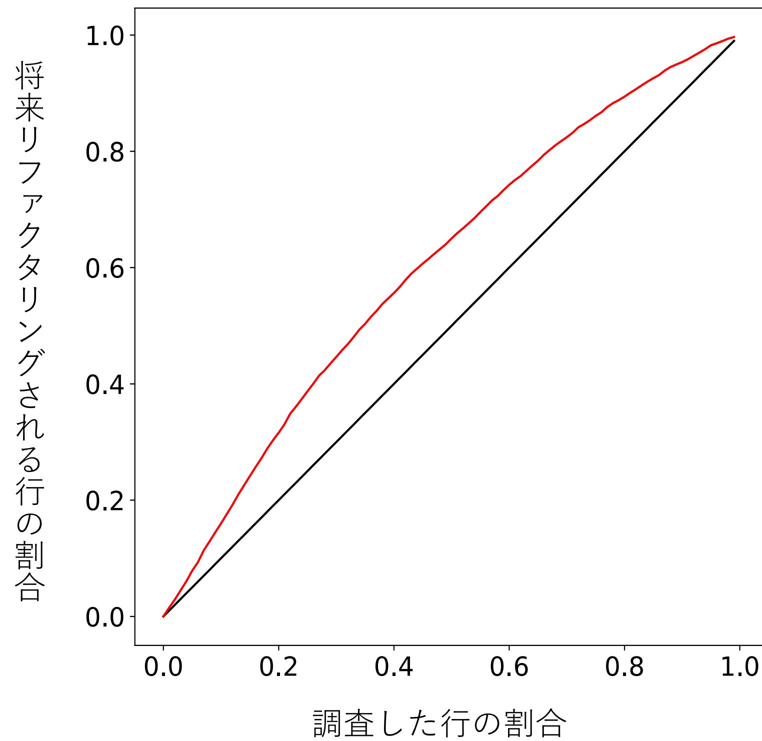


図 16　リファクタリングによって削除される行の自然さ

RQ3 の結論：リファクタリングすべき箇所の絞り込みに自然さは有用である．

4.4　まとめ

本章では自然さによるソースコードの評価指標をリファクタリングに適用するために 3 つのリサーチクエスチョンをたてた．リサーチクエスチョンを評価するために行った実験の結果，以下のことが分かった．

- リファクタリングによってソースコードの自然さが向上する傾向がある．
- リファクタリングの対象となるソースコードは他と比べて自然さが低い．
- リファクタリングすべき箇所の優先度付けに自然さが有効である．

この結果から，リファクタリング支援に対して自然さは有効であると言える．

今後の課題はまず，言語モデルを改良し，ソースコードの特徴をより正確にとらえることである．深層学習による言語モデルを用いた手法による成果が報告されており，もちいることでソースコードの評価に適した言語モデルを構築できる可能性がある．

また，自然さを用いたソースコード評価を IDE やコードレビューシステムに組み込んだツールを作成することで，自然さによるソースコード評価指標が開発支援に役に立つのか検証する必要がある．

5　妥当性の脅威

5.1　学習用ソースコードの品質

言語モデルを用いた自然さは，出現頻度が高いパターンであるほどより自然になるため，学習用ソースコードに保守性の低いソースコードが多く含まれている場合，品質の低いソースコードに対して自然であると出力してしまう可能性がある．

5.2　リファクタリングに関する実験

RQ1 における実験では，1 つのプロジェクトの 28 コミットしか評価されておらず，プロジェクトによる偏りが存在する可能性がある．また，リファクタリングが行われたかどうかは目視によって判別しているため誤りが含まれる可能性がある．

RQ2，RQ3 の実験で用いた行レベルデータセットの構築では，バグ導入コミット検出における SZZ アルゴリズム [17] と同様に，`git-blame` コマンドを用いてリファクタリングの原因となった変更を特定している．そのため，同じ場所に対して複数回の変更が加えられている場合，本当にリファクタリングの原因となったコミットを取得できない可能性がある [46]．

RQ3 の実験で用いた AUCEC という指標は，どれだけ効率的にリファクタリングで削除される行を見つけ出すことができるかという指標である．そのため偽陰性については考慮されず，指標が高くても開発者にとってはそれほど役に立たない可能性がある．

6 おわりに

本研究では，ソースコードを自然さという観点から評価するために，言語モデルを用いて計算される単語の生成確率を用いる方法によって，ソースコードを定量的に評価する手法を提案した．提案した評価手法をリファクタリング支援に適用した．

ソースコードの自然さを用いた評価指標がリファクタリングに対して有用であるかを評価するために，以下の 3 つのリサーチクエスチョンから評価を行った．

RQ1: リファクタリングによって自然さはどう変化するか

RQ2: リファクタリングによって削除される行の自然さは低いか

RQ3: リファクタリングすべき箇所の絞り込みに自然さは有用か

リサーチクエスチョンを評価するためにまず，オープンソースプロジェクトである JUnit4 に対して実際に行われたリファクタリングに対して，自然さが向上しているか，減少しているかを調査した．その結果，28 個のリファクタリングのうち 19 個のリファクタリングにおいて自然さが向上しており，リファクタリングを行うことによって自然さが向上する傾向があることがわかった．

次に，開発履歴のあるコミットの時点でのスナップショットに対して，どの行がリファクタリングによって削除されるかを SZZ アルゴリズムによって判定したデータセットを構築した．このデータセットをもとに，リファクタリングによって削除される行の自然さを調査した結果，そうでない行に比べて，中央値で 0.81 エントロピーが高く，自然さが低い傾向があることがわかった．

さらに，リファクタリングすべき行を自然さを用いて順位付けできるかどうかを調査し，AUCEC による評価指標が自然さを用いない場合に比べて，AUCEC による評価指標で，73% 高い性能を得た．このことから自然さはリファクタリングすべき箇所の絞り込みに有用であるといえる．

以上の調査の結果，リサーチクエスチョンへの回答は以下のようになった．

RQ1: リファクタリングによって自然さは向上する傾向がある

RQ2: リファクタリングによって削除される行の自然さは他と比べて低い

RQ3: リファクタリングすべき箇所の絞り込みに自然さは有用である

今後の課題としてはまず，言語モデルを改良し，ソースコードの特徴をより正確にとらえることである．深層学習による言語モデルを用いた手法による成果が報告されており，もちいることでソースコードの評価に適した言語モデルを構築できる可能性がある．

また，自然さを用いたソースコード評価を IDE やコードレビューシステムに組み込んだツールを作成することで，実際の開発者に対する被験者実験を行い，自然さによるソースコード評価指標が開発支

援に役に立つのか検証する必要がある．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，暖かく励まして頂きました楠本 真二 教授に心より感謝申し上げます．

本研究の全過程において，終始熱心なご指導を頂きました，肥後 芳樹 准教授に深く感謝申し上げます．

本研究に関して，的確で丁寧なご助言を頂きました，柗本 真佑 助教に心より感謝申し上げます．

本研究を進めるにあたり，様々な形で励まし，協力を頂いたその他の楠本研究室の皆様に深く感謝申し上げます．

最後に，本研究に至るまでに講義等でお世話になりました大阪大学大学院情報科学研究科の諸先生方に，この場を借りて心より御礼申し上げます．

参考文献

[1] Miltiadis Allamanis and Charles A. Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pp. 207–216, 2013.

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, Vol. 51, No. 4, pp. 81:1–81:37, 2018.

[3] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering*, pp. 837–847, 2012.

[4] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[5] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the Localness of Software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 269–280, 2014.

[6] Veselin Raychev, Martin T. Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 419–428, 2014.

[7] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning Natural Coding Conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 281–293, 2014.

[8] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. Syntax and sensibility: Using language models to detect and correct syntax errors. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*, pp. 311–322, 2018.

[9] Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. Syntax errors just aren't natural: improving error reporting with language models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 252–261, 2014.

[10] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering*, pp. 428–439, 2016.

[11] Vincent Hellendoorn, Premkumar T. Devanbu, and Alberto Bacchelli. Will they like this? evaluating code contributions with language models. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pp. 157–167, 2015.

[12] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, pp. 308–320, 1976.

[13] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp. 476–493, 1994.

[14] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1999.

[15] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics Based Refactoring. In *Proceedings of the 4th European Conference on Software Maintenance and Reengineering*, pp. 30–38, 2001.

[16] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, Vol. 107, pp. 1–14, 2015.

[17] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pp. 1–5, 2005.

[18] Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language*, Vol. 13, No. 4, pp. 359–393, 1999.

[19] Matthieu Jimenez, Maxime Cordy, Yves Le Traon, and Mike Papadakis. On the impact of tokenizer and parameters on n-gram based code analysis. In *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution*, pp. 437–448, 2018.

[20] Vincent Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 763–773, 2017.

[21] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *Journal of Machine Learning Research*, Vol. 3, pp. 1137–1155, 2003.

[22] Tomas Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association*, pp. 1045–1048, 2010.

[23] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*,

Vol. 9, No. 8, pp. 1735–1780, 1997.

[24] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. LSTM neural networks for language modeling. In *Proceedings of the 13th Annual Conference of the International Speech Communication Association*, pp. 194–197, 2012.

[25] Martin White, Christopher Vendome, Mario Linares Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pp. 334–345, 2015.

[26] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *Proceedings of the 6th International Conference on Learning Representations*, 2018.

[27] Anh Tuan Nguyen, Trong Duc Nguyen, Hung Dang Phan, and Tien N. Nguyen. A deep neural network language model with contexts for source code. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*, pp. 323–334, 2018.

[28] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. sk_p: a neural program corrector for moocs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pp. 39–40, 2016.

[29] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. Deepfix: Fixing common C language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pp. 1345–1351, 2017.

[30] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. Neuro-symbolic program corrector for introductory programming assignments. In *Proceedings of the 40th International Conference on Software Engineering*, pp. 60–70.

[31] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 2016.

[32] Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 135–146, 2017.

[33] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, Vol. 3, No.

POPL, pp. 40:1–40:29, 2019.

[34] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, Vol. 27, No. 1, pp. 1–12, 2001.

[35] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change history information. In *Proceedings of the 2013 28th IEEE/ACM International Conference on Automated Software Engineering*, pp. 268–278, 2013.

[36] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo da Silva Sousa, Rafael Maiani de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. Understanding the impact of refactoring on smells: a longitudinal study of 23 software projects. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pp. 465–475, 2017.

[37] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. The scent of a smell: An extensive comparison between textual and structural smells. *IEEE Transactions on Software Engineering*, Vol. 44, No. 10, pp. 977–1000, 2018.

[38] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 858–870, 2016.

[39] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. A textual-based technique for smell detection. In *Proceedings of the 24th IEEE International Conference on Program Comprehension*, pp. 1–10, 2016.

[40] Ayaka Imazato, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Finding extract method refactoring opportunities by analyzing development history. In *Proceedings of the 41st IEEE Annual Computer Software and Applications Conference*, pp. 190–195, 2017.

[41] Dario Di Nucci, Fabio Palomba, Damian A. Tamburri, Alexander Serebrenik, and Andrea De Lucia. Detecting code smells using machine learning techniques: Are we there yet? In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*, pp. 612–621, 2018.

[42] Nikolaos Tsantalis, Matin Mansouri, Laleh Mousavi Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, pp. 483–494, 2018.

[43] Yoshiki Higo and Shinji Kusumoto. How should we measure functional sameness from program source code? an exploratory study on java methods. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 294–305, 2014.

[44] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences.* Lawrence Erlbaum Associates, 1988.

[45] Erik Arisholm, Lionel C. Briand, and Eivind B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, Vol. 83, No. 1, pp. 2–17, 2010.

[46] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, Vol. 43, No. 7, pp. 641–657, 2017.