N-gram IDF を利用したソースコード内の特徴的部分抽出手法

小林 勇揮
京都工芸繊維大学 大学院工芸科学研究科 情報工学専攻
y-kobayashi@se.is.kit.ac.jp

水野 修
京都工芸繊維大学 情報工学・人間科学系
o-mizuno@kit.ac.jp

要旨

従来の大域的な語の重み付け手法である *IDF*（*Inverse Document Frequency*）には，単語 *N-gram* に対して適用できない欠点があった．しかし，近年の研究により，*IDF* を単語 *N-gram* に対して適用する手法が提案された．本研究では，この *N-gram IDF* をソースコードに対して適用し，ソースコード中の特徴的部分の抽出に応用できると考えた．具体的には，局所的重み付けである *TF*（*Term Frequency*）と *N-gram IDF* を利用した語の重み付け手法である $TF\text{-}IDF_{N-gram}$ を用いて，ソースコードごとの特徴語の抽出を行った．そして，その特徴語の行ごとの出現頻度を求めて，ソースコード中の特徴的部分の抽出を行った．まず，サンプルプログラムを用いて特徴語抽出の評価実験を行い，ソースコードにおいても特徴語をある程度抽出できることを示した．次に，*Apache Ant* の公開されているソースコードを用いて特徴的部分抽出を行い，またソースコードの変更による特徴的部分の変化についても調べた．その結果，ソースコードから特徴的部分の抽出をすることができた．また，その抽出した特徴的部分は，ソースコードの変更によってもソースコード全体の相対位置の変化が少ないことを示した．

1 はじめに

語の重み付け手法（term weighting scheme）はテキスト解析において重要な技術であり，情報検索や文書クラスタリングなど幅広い分野で利用されている．そこで，この語の重み付け手法をソースコードに対して適用し，ソースコード中の特徴的部分を特定することによって，開発者がソースコードに対してリファクタリングやメンテナンスをする際の役に立つのではないかと考えた．また語の重み付け手法では IDF（Inverse Document Frequency）という大域的な語の重み付け手法が利用されるが，この IDF には単語 N-gram に対して適用できない欠点があった．しかし，近年の研究により，この IDF を単語 N-gram に対して適用する手法が提案された [14]．

本研究では，この N-gram IDF を利用してソースコード中の特徴的部分を抽出する方法を提案する．この研究の動機としては，今までの IDF では一語の単語に対してしか語の重み付けができず，単語 N-gram に対して重み付けができなかった．IDF をソースコードに対して適用するときに，一語の場合はソースコード中でその語がどのような処理をするときに特徴的なのかが分かりづらい問題がある．例えば，ある語「name」の重みづけが大きくなったときに，その語が「int name」や「name = 0」など，宣言されたときが重要なのか，代入があったときが重要なのかが分からなかった．しかし，この N-gram IDF を利用すると，それを区別することができるため，ソースコードに対して利用すると有効性が得られると考えた．本研究の実施にあたっては，先行研究でのアルゴリズムの公開されている実装[1]を用いた．

2 準備

2.1 語の重み付け手法

語の重み付け手法（term weighting scheme）はテキスト解析において重要な技術である．その中で，代表的なものとして TF-IDF（Term Frequency-Inverse Document Frequency）がある．TF-IDF は，文書中に出現する特定の単語がどのくらい特徴的であるかを識別するための指標で，情報検索 [13]，文書クラスタリング [3]，特徴語抽出 [6]，映像中のオブジェクトマッチング [12] など，幅

[1] https://github.com/iwnsew/ngweight

広いアプリケーションで利用することができる．また，複数の観点 [2] [11] [7] から理論的説明が与えられており，多くの人が TF-IDF を利用する根拠となっている．本研究では，この語の重み付け手法をソースコードに対して利用することにした．次の節で，TF-IDF について詳しく説明していく．

2.2 TF-IDF

TF-IDF のような語の重み付け手法は，一般的に，局所的重み付けと大域的重み付けの二つの要素からなる．局所的重み付けは，ある文書に対する語の出現頻度から計算され，対象とする文書によって重みは変化する．一方，大域的重み付けは，文書集合全体における語の文書頻度から計算され，語の重みは対象とする文書によらず一定である．TF-IDF は具体的には，以下の式（1）により語 t の文書 $d \in D$ における重みを計算する．

$$TF\text{-}IDF(t,d) = tf(t,d) \cdot \log \frac{|D|}{df(t)} \tag{1}$$

ここで，$tf(t,d)$ は文書 d における語 t の出現頻度，$df(t)$ は文書集合 D における t の文書頻度，$|D|$ は D の文書数である．

この式（1）の中で，局所的重み付けは以下の式（2）で表される．

$$TF(t,d) = tf(t,d) \tag{2}$$

また，大域的重み付けは以下の式（3）で表される．

$$IDF(t) = \log \frac{|D|}{df(t)} \tag{3}$$

特に，大域的重み付けである IDF [8] は様々な語の重み付け手法で採用されている．IDF が様々な語の重み付け手法として採用されている理由として，式（3）のような簡潔さとロバスト性が挙げられる．実際に，IDF は様々な文献 [2] [11] [9] において理論的にロバストであることが示されている．

2.3 N-gram IDF

従来の IDF の欠点として，単語 N-gram に対して適用できないことがあった．単語 N-gram に対する IDF は，その連続する単語のつながりが不自然であるほど大きい重み付けをしてしまっていた．例えば，「Osaka University」と「Osaka be」では単語のつながりが不自然な「Osaka be」の方が文書に出現する割合が低くなるため，「Osaka be」の方が大きい重み付けをしてしまう．しかし，大阪大学大学院情報科学研究科の白川真澄らによる研究 [14] により，単語 N-gram に対しても IDF を適用することができることが発見された．この手法は，文字列が出現する Web ページをシャノン・ファノ符号によって表現したとき，空文字からの情報距離が対象の文字列の IDF となることを利用している．具体的には，以下の式（4）によって単語 N-gram g に対する IDF を計算する．

$$IDF_{N-gram}(g) = \log \frac{|D| \cdot df(g)}{df(\theta(g))^2} \tag{4}$$

ここで，$|D|$ は D の文書数，$df(g)$ は文書集合 D における g の文書頻度，$\theta(g)$ は g を構成する各単語の論理積，$df(\theta(g))$ は文書集合 D における $\theta(g)$ を満たす文書数である．また，各単語の論理積とは，文書集合において g を構成する各単語が含まれている文書である．

3 ソースコードに対する N-gram IDF の適用

ここではソースコードに対して N-gram IDF を使用するにあたって，どのような前処理をしたり，注意事項があるのか挙げる．

また，ソースコードに対してのみ N-gram IDF を使用したいので，ソースコード中のコメントがあるならそれを取り除き，その取り除いたソースコードに対し N-gram IDF を使用することにする．

3.1 単語の分割

ここではソースコードの単語の分け方について説明する．英語などのテキストは基本的に空文字によって単語を分割する．しかし，ソースコードを単語に分けるとき，プログラマや開発環境によって空文字の有無が発生する．例えば，「 a = b + c 」と「 a=b+c 」がある．このとき，前者は「a」「=」「b」「+」「c」の 5 文字で認識し，問題はない．一方，後者は空文字がないため，「a=b+c」の 1 文字で認識してしまうと問題が発生する．この問題を解決するために，ソースコードを単語に分ける際の規則は以下のようにする．実験では，C と Java のプログラミング言語に対して単語分けを行った．

- 基本的にスペース区切りで単語を分ける．
- 以下の記号は一単語としてみなす．
  ! ? ” ’ # $ % & | ( ) { } [ ] = < > + - * / \ ~ ^ @ : ; , .

これにより，ソースコードを単語に分けていく．

さらに詳しく説明をするために，この規則を適用してソースコードを単語に分けるときの一つの例を挙げる．例えば，次のようなソースコードがあるとする．

例）`name.name_get()=1+2;`

これを前述の規則に適用すると,「name」「.」「name_get」「(」「)」「=」「1」「+」「2」「;」の 10 単語に分かれる．ここで注意すべきところは「name_get」が 3 語に分かれないことである．これはソースコードにおいて記号の中でも「_」が変数の名前の一部として利用されるためである．特にこの「_」が変数の名前の一部として利用されていることが多いため，この記号を一語として見なさないことにした．また，Java などの言語では変数の名前の一部に「_」以外の記号を使うことがある．例えば,「$」や「@」,「%」などが挙げられる．しかし，そのような場合は先頭で始まったり，特徴的な使い方をされることが多いため 1 語として見なすことにした．

3.2 ソースコード中の特徴語判定方法

本研究では，ソースコード中の特徴的部分を発見するためにソースコード中の特徴語を判定する．そのために，語の大域的重み付けである N-gram IDF と語の局所的重み付けである TF（Term Frequency）を利用した $TF\text{-}IDF_{N-gram}$ を利用する．具体的には，式（2）と式（4）より，N-gram g のソースコード $d \in D$ における重みは以下の式（5）で表される．

$$TF\text{-}IDF_{N-gram}(g,d) = tf(g,d) \cdot \log \frac{|D| \cdot df(g)}{df(\theta(g))^2} \quad (5)$$

ここで，ソースコード $d \in D$ は，D をソースファイル群とし，その中の任意の一つのファイル中のソースコードを d とし，ソースファイル単位で分割する．よって，$tf(g,d)$ はソースコード d 中の g の出現回数，$df(g)$ は g が出現するソースファイル数，$df(\theta(g))$ は $\theta(g)$ を満たすソースファイル数である．

この式（5）を利用して，ソースコード中の特徴語を求めていく．以下に，その手順を示す．

1. 語の重み付けをするためのソースファイルを複数用意する．
2. それぞれのソースファイルのソースコード中のコメントを取り除く．
3. それぞれのソースコードを単語ごとに分割していく．
4. N-gram g とその語の長さ（N-gram の N）と $df(g)$, $df(\theta(g))$ を求める．
5. $df(g)$，$df(\theta(g))$ の値と式（4）を利用して，N-gram IDF を求める．
6. 求めた N-gram g からそれぞれのソースコードごとの $tf(g,d)$ を求める．
7. 5. と 6. の結果より，式（5）から $TF\text{-}IDF_{N-gram}(g,d)$ を求める．
8. 求めた $TF\text{-}IDF_{N-gram}(g,d)$ の値で N-gram g を降順に並び替え，その上位の g を特徴語とする．

ここで $df(\theta(g))$ の値を計算することは，単語の論理積に対する出現頻度を計算する必要があるため計算量が多い．また，N-gram g の種類が非常に多いため，ソースコード中から選ぶ N-gram g を絞ることも必要である．よって，ここでは白川真澄らの文字列解析手法を使った実装方法を用いた．まず，あらゆる N-gram g の中からある程度数を絞るために，拡張接尾辞配列 [1] を用いた極大部分文字列の抽出 [10] を行う．この極大部分文字列を N-gram g とする．次に $df(\theta(g))$ の値を計算するためにはウェーブレット木 [5] を利用することで計算量の短縮を行った．ウェーブレット木というデータ構造が文書頻度の計算処理においても高速に行えることは Gagie ら [4] が示している．

3.3 ソースコード中の特徴的部分抽出方法

本研究では，ソースコード中の特徴的部分を抽出するにあたって，ソースコードを行単位で確認し，特徴語が出現した行を数えていくことにした．そして，特徴語が出現した回数を縦軸に，ソースコードの行数を横軸に取ったヒストグラムを作成する．そして，ヒストグラムの値が高くなっている行に注目して，その部分を特徴的部分として判定する．

また，ソースコード中の特徴的部分を抽出するときにはソースコードを行単位で確認する．そのため，ソースコードを単語ごとに分割していくときには行ごとに単語に分割することにする．

3.4 研究設問

ここでは本研究の目的であるソースコード中の特徴的部分を抽出するにあたって，確かめたい研究設問につい

て説明する．

本研究で利用している N-gram IDF は元々テキスト文書の語の重み付けとして利用されているものである．そのため，それをソースコードに利用した第 3 章の節 3.2 の特徴語判定方法でソースコード中における特徴語を見つけることができるのかという問題がある．また，ソースコード中の特徴的部分を抽出する際に，見つかった特徴語が本当に特徴語である必要もある．よって，以下のような研究設問を設けた．

> RQ1. ソースコード中から特徴語を抽出できるのか

第 3 章の節 3.3 で説明した方法では，作成したヒストグラムを利用して，ソースコード中の特徴語が出現した回数が多い行を特徴的部分として抽出する．そのため，このヒストグラムを利用して，ソースコード中の特徴的部分の特定ができる必要がある．これを確かめるため，以下のような研究設問を設けた．

> RQ2. ソースコード中から特徴的部分を抽出できるのか

最後に，もし特徴的部分が抽出できるなら，その抽出した特徴的部分がソースコードの変更によってどのように変化するのかを調べたい．例えば，抽出した特徴的部分がソースコードの変更によって消えていったなら，その特徴的部分が何らかの変更が必要なソースコードである言えるかもしれない．また，抽出した特徴的部分がソースコードの変更によって消えずにそのまま存在するなら，大幅な変更を加える必要のないソースコードであると言えるかもしれない．したがって，そのような特徴が分かると開発者がソースコードに改良を加える際に何かの手助けになるかもしれない．よって，以下のような研究設問を設けた．

> RQ3. ソースコードの変更によって特徴的部分にはどのような変化があるのか

4 実験方法

第 3 章で述べた研究設問を調べるために，2 つの実験を行った．

```c
/* header files */
#include <stdio.h>
#include <stdlib.h>

/* main */
int main(void) {
 FILE *fp;
 char *filename = "sample.txt";
 int ch;

 /* ファイルのオープン */
 if ((fp = fopen(filename, "r")) == NULL) {
  fprintf(stderr, "%sのオープンに失敗しました.\n", filename);
  exit(EXIT_FAILURE);
 }

 /* ファイルの終端まで文字を読み取り表示する */
 while (( ch = fgetc(fp)) != EOF ) {
  putchar(ch);
 }

 /* ファイルのクローズ */
 fclose(fp);

 return EXIT_SUCCESS;
}
```

図 1. C 言語関数辞典 fgetc のサンプルプログラム[4]

まず，1 つ目の実験ではサンプルプログラムを用いてソースコード中の特徴語抽出の評価実験を行った．この実験結果を用いて RQ1 について答える．

次に，2 つ目の実験ではオープンソースプロジェクトを用いてソースコード中の特徴的部分抽出の実験を行った．この実験結果を用いて RQ2，RQ3 について答える．

4.1 サンプルプログラムを用いた特徴語抽出

「C 言語関数辞典[2]」の Web ページで公開されているサンプルプログラムを用いて特徴語抽出の評価実験を行った．C 言語関数辞典では，C 言語の関数やマクロの使い方に関する説明を主にしている．この Web ページでは，関数やマクロの説明がヘッダファイル別やアルファベット別などによって分類されて，掲載されている．また，関数の説明によってはサンプルプログラムを使って説明をしている場合がある．例えば，関数 fgetc ではサンプルプログラムを使って説明をしている．そのサンプルプログラムは図 1 である．このようなサンプルプログラムからコメントを除いて特徴語抽出を行う．また加えて，このサンプルプログラムでは文字列の出力の際に日本語が使われているので，特徴語を計算する前にすべての日本語を「JAPANESE」に変換した．

[2]http://www.c-tipsref.com/

このサンプルプログラムを複数用いて特徴語抽出の評価実験を行う．具体的には，C 言語関数辞典内のサンプルプログラムを集めてきて，それぞれのサンプルプログラムで説明している関数名をファイル名とし保存する．また，それぞれのサンプルプログラムのファイルの正解語はそのファイルで説明している「関数の名前」を含むものとする．例えば，図 1 のサンプルプログラムの場合，「fgetc」を含む単語 N-gram が正解語となる．さらに，サンプルプログラムは，Web ページ内のヘッダファイル一覧から「stdio.h」「stdlib.h」「string.h」「ctype.h」の 4 つのヘッダファイルを選択し，その中の関数の説明の際にサンプルプログラムを使用している箇所から集めた．集めたサンプルプログラムのファイル数は全部で 94 個になった．

評価実験では第 3 章の節 3.2 で説明したソースコード中の特徴語判定方法を用いて行う．それぞれのサンプルプログラム中の特徴語に順位付けをし，1 位の特徴語にそのサンプルプログラムの関数名が入っていたら正解として数え，その正解率を調べた．正解率 R は正解数を a とし，D をファイル数とすると，以下の式（6）で求める．

$$R = \frac{a}{D} \tag{6}$$

したがって，この正解率の値でソースコード中の特徴語抽出の評価実験を行う．

4.2 Apache Ant を用いた特徴的部分抽出

Apache ソフトウェア財団が開発しているオープンソースのソフトウェアプロジェクトである Apache Ant というビルドツールソフトウェアの Git リポジトリ[5]を用いて特徴的部分抽出の実験を行った．

また，ソースコードの変更による特徴的部分の変化についても調べるため，特定のファイルの変更があった際の N-gram IDF を全て求める．具体的には，以下のようにして特定のファイル（以下ファイル A とする）の変更時のすべての N-gram IDF を求めることにする．

1. 現時点[6]の Apache Ant のソースコードを全て集めてきて，一つのファイル（以下ファイル B とする）にする．また，ファイル B を作成する際は集めてきたソースコードの参照先がわかるように一つ一つ区切る．
2. Apache Ant の全コミットのログを集めてきて，ソースコードファイルに対するすべての変更履歴を調べる．
3. 変更履歴を探索していき，ファイル B のソースコードの一部をその変更があったファイルのソースコードに置き換えていく．もし，変更があったファイルがファイル A なら，ファイル B を書き換えた後に N-gram IDF を求める．

以降，3. を繰り返すことによって，ファイル A の N-gram IDF を全て求めることができる．

この求めた N-gram IDF を用いて，$TF\text{-}IDF_{N-gram}$ をそれぞれのファイルについて求めて，行ごとの特徴語の出現頻度を表したヒストグラムを作成していく．また，ソースコードの変更による特徴的部分の変化については，横軸がソースコードの行数，縦軸が変更回数のヒートマップを作成し，特徴語の出現回数によって行ごとに色を変化させる．そして，その特徴的部分の色の変化によって，ソースコードの変更による特徴的部分の変化について調べることにする．

5 実験結果

5.1 RQ1:ソースコード中から特徴語を抽出できるのか

サンプルプログラムを用いた特徴語抽出の実験を利用して，RQ1 に答えていく．例えば，図 1 に対して特徴語抽出を行い，語の長さが 2 語以上で上位 10 位以内の単語 N-gram を表 1 に示す．表 1 により，最も特徴的な単語 N-gram が「 (ch 」となっていて，次いで「 (fp 」，その後に 4 つの単語 N-gram が同じ特徴度になっていることがわかる．さらに，このソースコードの正解の特徴語は「 `fgetc` 」を含むものとしていたので，それを含む単語 N-gram「 ch = fgetc (fp) 」は第 3 位となっている．このようにして，C 言語関数辞典から集めてきた全 94 個のソースコードのファイルに対して特徴語の抽出を行い，評価を行った．

まず，全てのサンプルプログラムに対して行った特徴語抽出の評価実験の結果をまとめた表が表 2 である．ここで正解数とは，単語 N-gram を順位付けしたときの 1 位の特徴語が正解の語を含んでいるサンプルプログラム

[4]参照元 http://www.c-tipsref.com/reference/stdio/fgetc.html (参照日 2017/02/01)
[5]https://github.com/apache/ant
[6]実験では 2016 年 10 月 18 日時点

表 1. fgetc のサンプルプログラムに対する特徴語抽出の結果

語の長さ	$tf(g,d)$	$df(g)$	$df(\theta(g))$	IDF_{N-gram}	$TF\text{-}IDF_{N-gram}$	単語 N-gram g
2	2	5	5	4.233	8.465	(ch
2	3	20	20	2.233	6.698	(fp
2	1	3	3	4.970	4.970	putchar (
5	1	3	3	4.970	4.970	while ((ch =
6	1	3	3	4.970	4.970	ch = fgetc (fp)
4	1	3	3	4.970	4.970	) { putchar (
6	1	4	4	4.555	4.555	) ! = EOF) {
2	2	20	20	2.233	4.465	fp)
3	2	20	20	2.233	4.465	(fp)
7	1	2	3	4.385	4.385	; } while ((ch =

表 2. ファイル数 94 個に対する特徴語の正解率

語の長さ N 語 [以上]	正解数	正解率
1	16	0.170
2	23	0.245
3	30	0.319
4	27	0.287
5	19	0.202
6	12	0.128
7	8	0.085
8	4	0.043
9	0	0.000

表 3. ファイル数 94 個に対する特徴語のヒット数とその適合率

語の長さ N 語 [以上]	ヒット数	正解数	適合率
1	50	16	0.320
2	47	23	0.489
3	44	30	0.682
4	39	27	0.692
5	29	19	0.655
6	14	12	0.857
7	10	8	0.800
8	7	4	0.571
9	0	0	0.000

の数である．また，正解率は式（6）を用いて計算している．表 2 より，正解率は長さ 3 語以上のときに最も高くなっていることがわかる．しかし，正解率自体は 0.319 とあまり高い値にはならなかった．そこで，サンプルプログラムに対して特徴語抽出を行った際のヒット数と適合率についても調べ，表 3 に示す．ここで，適合率 R-Prec は，ヒット数を h，正解数を a とすると以下の式（7）で表す．

$$R\text{-}Prec = \begin{cases} \frac{a}{h} & (\mathrm{h} > 0) \\ 0 & (\mathrm{h} = 0) \end{cases} \tag{7}$$

ヒット数 h とは，ファイル中のソースコードの単語 N-gram を順位付けした際に，その中に順位は関係なく正解語が入ってるファイルの数である．表 3 より，全ファイル 94 個に対して正解語を含む単語 N-gram が入ってたファイルの数は最大で 50 となっていて，約半数近くのファイルがヒットしていないことがわかる．さらに，語の長さが増えていくたびに値が低くなっていき，語の長さが 9 語以上で 0 になる．このことから，正解率が低かったのはプログラムが単語 N-gram を作成する際に，その単語 N-gram に正解の語が含んでいないものが約半数以上あったからと推測した．そのため，ソースコードの特徴を考えたテキストとは違う単語 N-gram の作成の必要があると考える．

また，適合率は語の長さが 3 語以上から 7 語以上までで 6 割を超え，6 語以上で最大値 0.857 をとる．よって，語の長さが 3 語以上の連続した語に対して特徴語の抽出を行うと，正解率が 0.319 で，適合率が 0.682 となり，どちらの値も比較的高い値になる．このことから，語の長さが 3 語以上の連続した語に対して特徴語の抽出を行い順位付けをすると，上位の語がソースコードにおいて特徴語である可能性が高くなることがわかる．

以上の結果より，ソースコード中から 3 語以上の連続した語なら，比較的高い確率で特徴語を抽出できるといえる．

5.2 RQ2:ソースコード中から特徴的部分を抽出できるのか

ここでは，Apache Ant を用いた特徴的部分抽出の実験を利用して，RQ2 に答えていく．

はじめに，RQ1 の実験の結果を利用して，語の長さが 3 語以上の単語 N-gram に対してのみ特徴語を求め，ソースコードのファイルを代表する特徴語とした． さらに，$TF\text{-}IDF_{N-gram}$ の値で並び替えた上位の単語 N-gram を

特徴語とする時に，上位 100 位以内の単語 N-gram を特徴語として候補を絞り込むこととした．

また，事前の実験で上位 100 位以内の単語 N-gram を特徴語とした結果，同じ行に複数の特徴語が出現することが分かった．行あたりの特徴語の数によって特徴的部分を特定する際に，ある一部の行の特徴語の数が大きくなると，その行以外の値が相対的に低くなり特定が難しくなる． 同じ行に複数の特徴語が出現する理由としては，特徴語の中に似た語の組み合わせで，かつ $TF\text{-}IDF_{N-gram}$ の値が近いからである．例えば，表 4 のような場合がある．この表は「 elementAt (i) 」と「 . elementAt (i) 」が特徴語となっていて，残りの 4 つは 2 つの特徴語を含む単語 N-gram を幾つか挙げたものである．この場合，表の出現箇所を見ると，2 つの特徴語の出現箇所が重なってしまっていることがわかる．この問題に対処するために，特徴語の出現箇所からそれ以降の順位の特徴語を含む単語 N-gram の出現箇所を除くことにした．これを適用すると，2 つの特徴語の出現箇所はそれぞれ「 elementAt (i) 」のとき出現箇所はなくなり，「 . elementAt (i) 」のときの出現箇所は 618 と 840 になる．これ以降，特徴語の出現箇所は以上のようにして求める．

次に，Apache Ant のどのソースコードに対して特徴的部分の抽出を行うか決める．2016 年 10 月 18 日時点で，Apache Ant では全 1222 個の拡張子が java のソースコードファイルが存在する．そのため，ある程度特徴量の高い可能性があるソースコードに絞って実験を行いたい．よって，2016 年 10 月 18 日時点の Apache Ant の全 1222 個のソースコードファイルに対して特徴語抽出を行い，大域的重み付け N-gram IDF の高い値が比較的多いものを探し出した．その結果の比較的多かったファイルの幾つかを表 5 に示した．この中から,「Main.java」と「Javadoc.java」を選んで実験を行った．

まず，2013 年 7 月 17 日時点の「Main.java」に対して特徴的部分抽出を行った結果について説明する．図 2 は特徴語の出現箇所を行ごとに数えていき，行ごとの特徴語の出現頻度をグラフにしたヒストグラムである．グラフから特徴的部分を値が 10 以上のグラフに注目して考えていく，まず 660 行から 700 行あたりが一番高くなり，次に 150 行から 200 行あたり，270 行から 300 行あたりが高くなっている．また，370 行から 400 行あたりも少し高くなっている．

一番値が高くなった 660 行から 700 行あたりは，

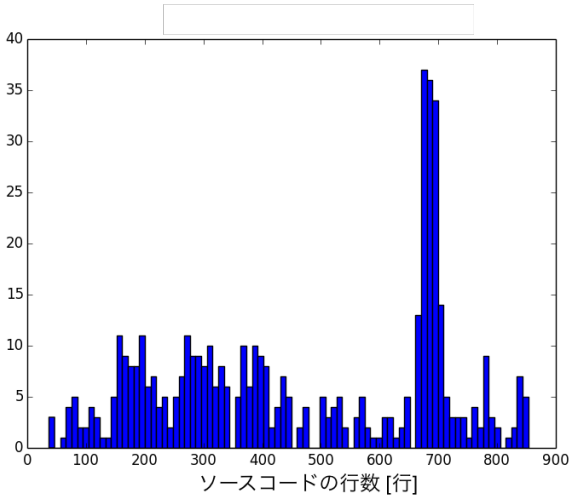

図 2. 2013 年 7 月 17 日における Main.java における特徴語の出現頻度

「Main.java」のソースコードでも見た目からも特徴的なものになっていた．それは，プログラムの引数の使用方法の文字列の出力をする関数（関数名 printUsage）であった．そのため，文字列を表示させるための関数を複数使用していた．次に値が高くなっていた 150 行から 200 行あたりや 270 行から 300 行あたりは「Main.java」のソースコードでは実行時に与えられた引数の処理をする関数（関数名 processArgs）になっていた．さらに詳しく見ていくと，150 行から 200 行あたりは引数の条件分岐のコードとなっていて，270 行から 300 行あたりはビルドファイルがなかったときの条件分岐のコードになっていた．また，次いで値が高くなっていた 370 行から 400 行あたりは，与えられた引数の処理をする際のそれぞれの引数で指定された値を変数に保存したり，その値が無効なものならエラー処理をする関数がいくつか並んでいた．これらの関数は関数 processArgs 内の 150 行から 200 行あたりの引数の条件分岐のコードで利用されていた．

よって,「Main.java」において特徴的部分は大きく分けて 3 つ挙げられる．

- 1 つ目は，660 行から 700 行あたりのプログラムの引数の使用方法の文字列を出力する関数 printUsage
- 2 つ目は，関数 processArgs で具体的には 150 行から 200 行あたりの引数の条件分岐のコードと 270 行から 300 行あたりはビルドファイルがなかったときの条件分岐のコード

表 4. 同じ行に複数の特徴語が出現する理由

語の長さ	$TF\text{-}IDF_{N-gram}$	単語 N-gram g	出現箇所 [行]
4	25.963	elementAt (i)	618,812,832,835,836,840
5	25.769	. elementAt (i)	618,812,832,835,836,840
6	16.942	names . elementAt (i)	812,832,835
8	11.295	(names . elementAt (i))	812,832
6	8.227	. elementAt (i))	812,832,836
6	0.835	. elementAt (i) .	835

表 5. 大域的重み付け N-gram IDF の高い値が多かったファイル

ファイル名	参照先
Javadoc.java	ant/src/main/org/apache/tools/ant/taskdefs/
BlockSort.java	ant/src/main/org/apache/tools/bzip2/
ZipOutputStream.java	ant/src/main/org/apache/tools/zip/
CBZip2InputStream.java	ant/src/main/org/apache/tools/bzip2/
CBZip2OutputStream.java	ant/src/main/org/apache/tools/bzip2/
IntrospectionHelper.java	ant/src/main/org/apache/tools/ant/
Zip.java	ant/src/main/org/apache/tools/ant/taskdefs/
ZipFile.java	ant/src/main/org/apache/tools/zip/
Main.java	ant/src/main/org/apache/tools/ant/

- 3 つ目は，370 行から 400 行あたりの与えられた引数の処理をする際のそれぞれの引数で指定された値を変数に保存したり，その値が無効なものならエラー処理をする関数群

また，2013 年 10 月 27 日時点の「Javadoc.java」に対して特徴的部分抽出を行った結果，図 3 に示すヒストグラムを得ることができた．これに基づき，次のように特徴的部分を抽出できた．

- 1 つ目は，50 行から 710 行あたりの「cmd.createArgument(」というコードを含む関数群
- 2 つ目は，関数 execute 内の 760 行あたりで関数呼び出しをしているソースコード
- 3 つ目は，870 行から 940 行あたりと 1050 行から 1200 行あたりの関数 execute を実行するために何らかの引数に値をセットする関数群

以上より，「Main.java」と「Javadoc.java」のソースコードについて，出力したヒストグラムから特徴語の出現頻度の値が高くなっている部分と低くなっている部分が存在し，その値が高くなっている部分の周辺を特徴的部分として抽出することができた．

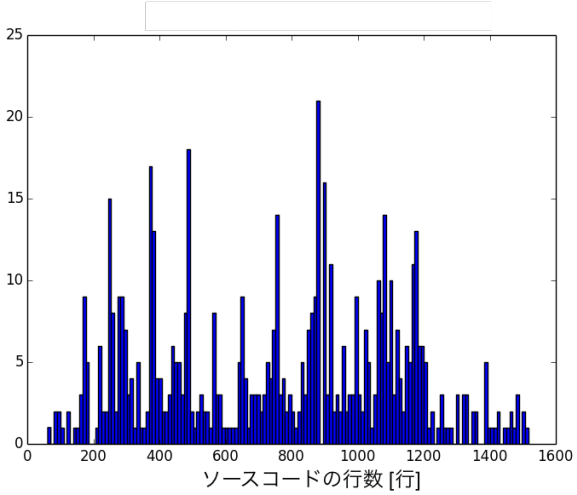

図 3. 2013 年 10 月 27 日における Javadoc.java における特徴語の出現頻度

5.3 RQ3:ソースコードの変更によって特徴的部分にはどのような変化があるのか

Apache Ant を用いた特徴的部分抽出の実験を利用して，RQ3 に答えていく．この実験で求めた「Main.java」と「Javadoc.java」の変更時のそれぞれの特徴語を用いて，10 行ごとの特徴語の出現回数を求めた．その値をヒートマップの値として，横軸を行数，縦軸を変更回数としてヒートマップを作成した．

まず，「Main.java」の全変更時に対して特徴的部分抽出を行い 10 行ごとの特徴語の頻出度をヒートマップにした結果について説明する．その結果を図にしたのが図 4 である．このソースコードの変更回数は 167 回で，行数は最大で 870 行であった．一番上が初期の古いソースコードの特徴量の分布を表していて，下に行くにつれて新しいソースコードの分布になっていく．図 4 を見ると，ソースコードの行数が徐々に増えていき，頻繁に変更がされていることがわかる．図 4 より，5.2 節で説明した

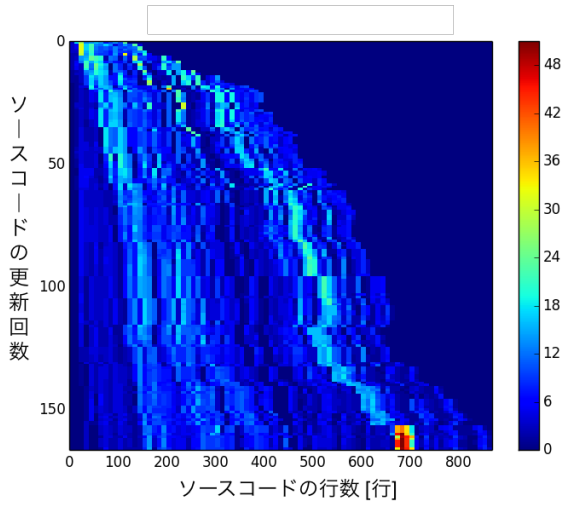

図 4. Main.java における特徴語出現頻度の時系列的推移

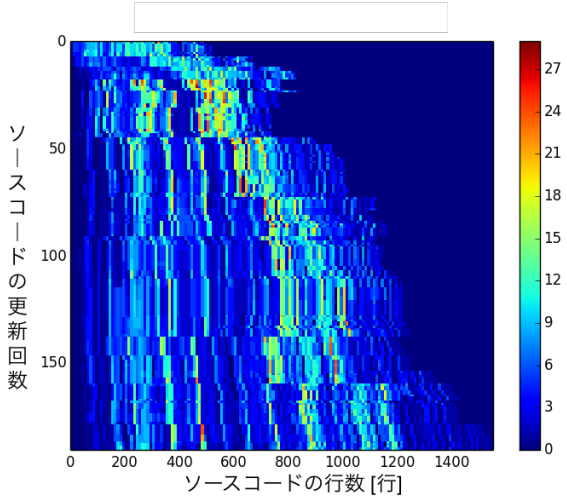

図 5. Javadoc.java における特徴語出現頻度の時系列的推移

特徴的部分の値が高くなっていることがわかり，それぞれのソースコードの特徴的部分はソースコードの変更があってもソースコード全体の相対的な特徴的部分の位置の変化は少ないことがわかる．

同じく,「Javadoc.java」の全変更時に対して特徴的部分抽出を行い 10 行ごとの特徴語の頻出度をヒートマップにした結果 (図 5) について説明する．このソースコードの変更回数は 191 回で，行数は最大で 1552 行であった．図より,「Javadoc.java」においてもソースコードの行数が徐々に増えていき，頻繁に変更がされていることがわかる．また，5.2 節で説明した特徴的部分の値が高くなっていることがわかり，それぞれのソースコードの特徴的部分もソースコードの変更があってもソースコード全体の相対的な特徴的部分の位置の変化が少ないことがわかる．この「Javadoc.java」においては更新回数が 50 回あたりなどでソースコードが大幅に変更があっても，それに合わせて特徴的部分がソースコード全体の位置に合わせて移動している．このことからも，特徴的部分の位置の変化が少ないことがわかる．以上より，ソースコードの変更によって特徴的部分の位置が他に移ることはなく，ソースコード全体の相対的な特徴的部分の位置の変化が少ないことがわかった．

6 まとめ

本研究では，単語 N-gram に対して重み付けができる N-gram IDF をソースコードに対して利用してソースコード中の特徴的部分の抽出を行った．具体的には N-gram IDF を利用した語の重み付け手法である $TF\text{-}IDF_{N-gram}$ を用いて，ソースコードごとの特徴語の抽出を行い，その特徴語の行ごとの出現頻度を求めて特徴的部分の抽出を行った．

適用実験の結果より，N-gram IDF を用いた語の重み付け手法である $TF\text{-}IDF_{N-gram}$ を利用すると，ソースコード中の特徴語を抽出でき，その特徴語の行ごとの出現頻度から特徴的部分を抽出することができた．また，その特徴的部分はソースコードの変更によってソースコード全体の相対位置の変化が少ないということがわかった．

今後の課題としてはサンプルプログラムを用いて特徴語抽出を行った際の，単語 N-gram の特徴語候補のヒット数の改善がある．今回の研究では，複数ある単語 N-gram の数を絞るために拡張接尾辞配列を用いた極大部分文字列の抽出を利用している．しかし，この単語 N-gram の選択方法ではヒット数が少なかったことから，ソースコードに対して有効な単語 N-gram を選択しないのかもしれない．よって，異なるソースコードの特徴を利用した単語 N-gram の選択方法が考えられる．

参考文献

[1] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. In *Journal of Discrete Algorithms*, Vol. 2, pp. 53–86, 3 2004.

[2] A. Aizawa. An information-theoretic perspective of tf-idf measures. In *Information Processing and Management*, Vol. 39, pp. 45–65, 2003.

[3] B.C. Fung, K. Wang, and M. Ester. Hierarchical document clustering using frequent itemsets. In *Proceedings of SIAM International Conference on Data Mining(SDM)*, pp. 59–70, 5 2003.

[4] T. Gagie, G. Navarro, and S.J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. In *Theoretical Computer Science*, Vol. 426-427, pp. 25–41, 4 2012.

[5] R. Grossi, A. Gupta, and J.S. Vitter. High-order entropy-compressed text indexes. In *SODA*, pp. 841–850, 2003.

[6] K.S. Hasan and V. Ng. Conundrums in unsupervised keyphrase extraction: Making sense of th state-of-the-art. In *Coling*, pp. 365–373, 8 2010.

[7] D. Hiemstra. A probabilistic justification for using tfx-idf term weighting in information retrieval. In *International Journal on Dgital Libraries*, Vol. 3, pp. 131–139, 9 2000.

[8] K.S. Jones. A statistical interpretation of term specificity and its application in retrieval. In *Journal of Documentation*, Vol. 28, pp. 11–21, 1972.

[9] D. Metzler. Generalized inverse document frequency. In *CIKM*, pp. 26–30, 10 2008.

[10] D. Okanohara and J. Tsujii. Text categorization with all substring features. In *SDM*, pp. 838–846, 4 2009.

[11] S. Robertson. Understanding inverse document frequency: On theoretical arguments for idf. In *Journal of Documentation*, Vol. 60, pp. 503–520, 2004.

[12] J. Sivic and A. Zisserman. Video google: A text retrieval approach to object matching in videos. In *ICCV*, pp. 1470–1477, 10 2003.

[13] H.C Wu, R.W.P. Luk, K.F Wong, and K.L. Kwok. Interpreting tf-idf term weights as making relevance decisions. In *ACM Transactions on Information Systems*, Vol. 26, pp. 13:1–13:37, 6 2008.

[14] 白川真澄, 原隆浩, 西尾章治郎. コルモゴロフ複雑性に基づく idf の単語 n-gram への適用. In *DEIM Forum*, 第 A 巻, pp. 3–5, 2015.