卒 業 研 究 報 告 書

題　目　　N-gram IDF を利用した
ソースコード中の重要部分抽出方法

指導教員　　水野　修　教授

京都工芸繊維大学 工芸科学部 情報工学課程

学生番号　　13122021

氏　　名　　小林　勇揮

平成 29 年 2 月 14 日提出

N-gram IDF を利用したソースコード中の重要部分抽出方法

平成 29 年 2 月 14 日　　13122021　小林　勇揮

概　要

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，ソースコード中の重要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重要語の抽出を行った．そして，その重要語の行ごとの出現頻度を求めて，ソースコード中の重要部分の抽出を行った．まず，サンプルプログラムを用いて重要語抽出の評価実験を行い，ソースコードにおいても重要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重要部分抽出を行い，またソースコードの変更による重要部分の変化についても調べた．その結果，ソースコードから重要部分の抽出をすることができた．また，その抽出した重要部分は，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．

目　次

1. 緒言

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，ソースコード中の重要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．

本研究では，この N-gram IDF を利用してソースコード中の重要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重要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(注 1)を用いた．

本報告の以降の構成は次のとおりである．まず，第 2 章で語の重み付け手法について詳しく説明する．次に，第 3 章でソースコードに対して N-gram IDF が使用できるようにプログラムを改良した点を説明する．また，ソースコードから重要語や重要部分を求めるための方法について説明する．そして，ソースコードに対して N-gram IDF を利用するとソースコード中の重要部分が見つかるのかという疑問に対して，具体的に 3 つの研究設問に分けて紹介する．次に，第 4 章でそれぞれの研究設問に答えるための実験方法を説明し，第 5 章でそれらの実験結果について説明する．最後に，実験方法や実験結果の概要や今後の課題について簡潔に説明する．

（注 1）: http://iwsew.com

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 [1]，文書クラスタリング [2]，特徴語抽出 [3]，映像中のオブジェクトマッチング [4] など，幅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 [5][6][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（2.1）により語 t の文書 $d \in D$ における重みを計算する．

$$TF\text{-}IDF(t,d) = tf(t,d) \cdot \log \frac{|D|}{df(t)} \tag{2.1}$$

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（2.1）の中で，局所的重み付けは以下の式（2.2）で表される．

$$TF(t,d) = tf(t,d) \tag{2.2}$$

また，大域的重み付けは以下の式（2.3）で表される．

$$IDF(t) = \log \frac{|D|}{df(t)} \tag{2.3}$$

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（2.3）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 [5][6][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．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 [10] 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（2.4）によって単語 N-gram g に対する IDF を計算する．

$$IDF_{N-gram}(g) = \log \frac{|D| \cdot df(g)}{df(\theta(g))^2} \tag{2.4}$$

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．

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．

- 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 や Ruby 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 ソースコード中の重要語判定方法

本研究では，ソースコード中の重要部分を発見するためにソースコード中の重要語を判定する．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（2.2）と式（2.4）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（3.1）で表される．

$$TF\text{-}IDF_{N-gram}(g,d) = tf(g,d) \cdot \log \frac{|D| \cdot df(g)}{df(\theta(g))^2} \tag{3.1}$$

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（3.1）を利用して，ソースコード中の重要語を求めていく．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（2.4）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（3.1）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 を重要語とする．

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 Ngram 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 [11] を用いた極大部分文字列の抽出 [12] 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 [13] 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 [14] が示している．

3.3 ソースコード中の重要部分抽出方法

本研究では，ソースコード中の重要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，重要語が出現した行を数えていくことにした．そして，重要語が出現した行を利用して，重要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．また，行数にはコメントを含めない．作成したヒストグラムを用いて，ソースコード中の重要部分を抽出する．具体的には，ヒストグラムの値が高くなっている行に注目して，その部分を重要部分として判定する．

また，ソースコード中の重要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重要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．

まず，ソースコード中から重要語を抽出できるのかを調べたい．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 の重要語判定方法でソースコード中における重要語を見つけることができるのかという問題がある．また，ソースコード中の重要部分を抽出する際に，見つかった重要語が本当に重要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重要語を抽出できるのか

次に，ソースコード中から重要部分が抽出できるのかを調べたい．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，ソースコード中の重要語が出現した回数が多い行を重要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，ソースコード中の重要部分の特定ができる必要がある．

よって，以下のような研究設問を設けた．

RQ2. ソースコード中から重要部分を抽出できるのか

最後に，もし重要部分が抽出できるなら，その抽出した重要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．ソースコードの変更によって重要部分が変化していくなら，その重要部分の特徴がわかるかもしれない．例えば，抽出した重要部分がソースコードの変更によって消えていったなら，その重要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，抽出した重要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重要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．

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重要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重要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重要語抽出

C 言語関数辞典 (注 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重要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 4.1 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重要語抽出を行う．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，重要語を計算する前にすべての日本語を「JAPANESE」に変換した．

よって，このサンプルプログラムを複数用いて重要語抽出の評価実験を行うことにした．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した．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 4.1 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．

(注 2) : http://www.c-tipsref.com/
(注 3) : 参照元 http://www.c-tipsref.com/reference/stdio/fgetc.html (参照日 2017/02/01)

```
/* 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 4.1　C 言語関数辞典 **fgetc** のサンプルプログラム [注 3)]

評価実験では第 2 章で説明したソースコード中の重要語抽出方法を用いて行う．それぞれのサンプルプログラム中の重要語に順位付けをし，1 位の重要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（4.1）で求める．

$$R = \frac{a}{D} \tag{4.1}$$

したがって，この正解率の値でソースコード中の重要語抽出の評価実験を行う．

4.2 Apache Ant を用いた重要部分抽出

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(注 4)を用いて重要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重要部分の変化についても調べたい．そのために，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(注 5)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，行ごとの重要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，

（注 4）: https://github.com/apache/ant
（注 5）: 実験では 2016 年 10 月 18 日時点

ソースコードの変更による重要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，重要語の出現回数によって行ごとに色を変化させる．そして，その重要部分の色の変化によって，ソースコードの変更による重要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重要語を抽出できるのか

サンプルプログラムを用いた重要語抽出の実験を利用して，RQ1 に答えていく．例えば，図 4.1 に対して重要語抽出を行った結果は表 5.1 のようになる．小数点 4 桁以下は四捨五入している（以下同様）．また，語の長さが 2 語以上で上位 10 位以内の単語 N-gram を表にしている．表 5.1 により，最も重要な単語 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 が同じ重要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重要語は「 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 言語関数時点から集めてきた全 94 個のソースコードのファイルに対して重要語の抽出を行い，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重要語抽出の評価実験の結果をまとめた表が表 5.2 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 位の重要語が正解の語を含んでいるサンプルプログラムの数である．また，正解率は式（4.1）を用いて計算している．表 5.2 より，正解率は長さ 3 語以上のときに一番高くなっていることがわかる．しかし，その値が 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重要語抽出を行った際のヒット数と適合率を調べた．上位 10 位以内その表が表 5.3 である．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（5.1）で表す(注 6)．

$$R\text{-}Prec = \frac{a}{h} \tag{5.1}$$

このヒット数とは，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 5.3 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

(注 6) : ヒット数が 0 のときは適合率を 0 にした

表 5.1　fgetc のサンプルプログラムに対する重要語抽出の結果

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 5.2　ファイル数 94 個に対する重要語の正解率

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 **5.3**　ファイル数 **94** 個に対する重要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重要語の抽出を行うと，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重要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重要語である可能性が高くなることがわかる．

したがって，ソースコード中から 3 語以上の連続した語なら，比較的高い確率で重要語を抽出できるといえる．

5.2 RQ2:ソースコード中から重要部分を抽出できるのか

Apache Ant を用いた重要部分抽出の実験を利用して，RQ2 に答えていく．まずはじめに，重要部分を抽出する際にそれぞれのソースコードのファイルの重要語を求める必要があるのだが，そのときに 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 に対して重要語を求めるようにした．さらに重要語を求めるとき，$TF\text{-}IDF_{N-gram}$ の値で並び替えた上位の単語 N-gram を重要語とするのだが，この実験では重要箇所を求めるためにある程度の重要語の候補が欲しいので，上位 100 位以内の単語 N-gram を重要語とした．

ここで，事前の実験で上位 100 位以内の単語 N-gram を重要語とした結果，同じ行に複数の重要語が出現した．重要部分を特定する際には行あたりの重要語の数によって特定するのだが，ある一部の行の重要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重要部分の特定が難しくなる．同じ行に複数の重要語が出現する理由としては，重要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 5.4 のような場合がある．この表は「 elementAt (i) 」と「 . elementAt (i) 」が重要語となっていて，残りの 4 つは 2 つの重要語を含む単語 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 つの重要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，

表 **5.4**　同じ行に複数の重要語が出現する理由

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

重要語の出現箇所からそれ以降の順位の重要語を含む単語 N-gram の出現箇所を除くことにした．これを適用すると，2 つの重要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，重要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重要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，ある程度重要度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重要語抽出を行い，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 5.5 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」に対して重要部分抽出を行った結果について説明する．図 5.1 は重要語の出現箇所を行ごとに数えていき，行ごとの重要語の出現頻度をグラフにしたヒストグラムである．グラフから重要部分を値が 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,「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，次に少し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．

表 5.5　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/

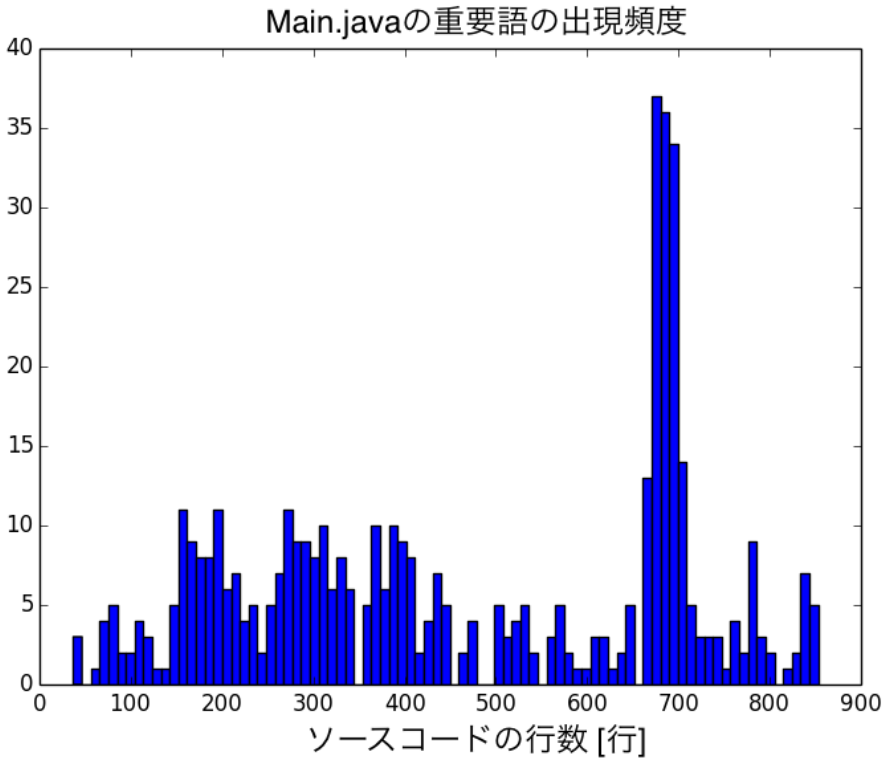

図 5.1　2013 年 7 月 17 日における Main.java のヒストグラム

よって,「Main.java」において重要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
- 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

次に，2013 年 10 月 27 日時点の「Javadoc.java」に対して重要部分抽出を行った結果について説明する．図 5.2 も「Main.java」と同様に重要語の出現箇所を行ごとに数えていき，行ごとの重要語の出現頻度をグラフにしたヒストグラムである．グラフから重要部分を値が 10 以上のグラフに注目して考えていく，まず 870 行から 940 行あたりが一番値が高くなり，次に 470 行から 490 行あたり，370 行から 390 行あたり，240 行から 250 行あたりと続いていく．また，760 行あたりも少し高くなっていて，1050 行から 1200 行あたりも他と比べて平均的に値が高くなっていることがわかる．

まず，一番値が高くなった 870 行から 940 行あたりは,「Javadoc.java」のソースコードでは generalJavadocArguments 関数や doDoclet 関数があり，それぞれの関数のソースコードを見るとどちらにも「 toExecute.createArgument().setValue(」というコードが頻出している．実際に，このソースコードの重要語には「toExecute」や「createArgument」,「setValue」を含む単語 N-gram が上位の方に入っていた．次に，470 行から 490 行と 370 行から 390 行あたり，240 行から 250 行あたりは「Javadoc.java」のソースコードでは「 cmd.createArgument(」というコードが頻出していた．そのために重要度が高くなったと考える．この「Javadoc.java」のソースコードの 50 行から 710 行あたりまでは比較的 getter や setter の関数などが並んでいる．その中でも特に「 cmd.createArgument(」というコードを含んでいる setter などの関数がある行のヒストグラムの値が高くなっていた．そのため，50 行から 710 行あたりでは，高い値が複数密集せずに一本だけ値が高くなったり，急に値が低くなったりしている．ゆえに，50 行から 710 行あたりの「 cmd.createArgument(」というコードを含んで

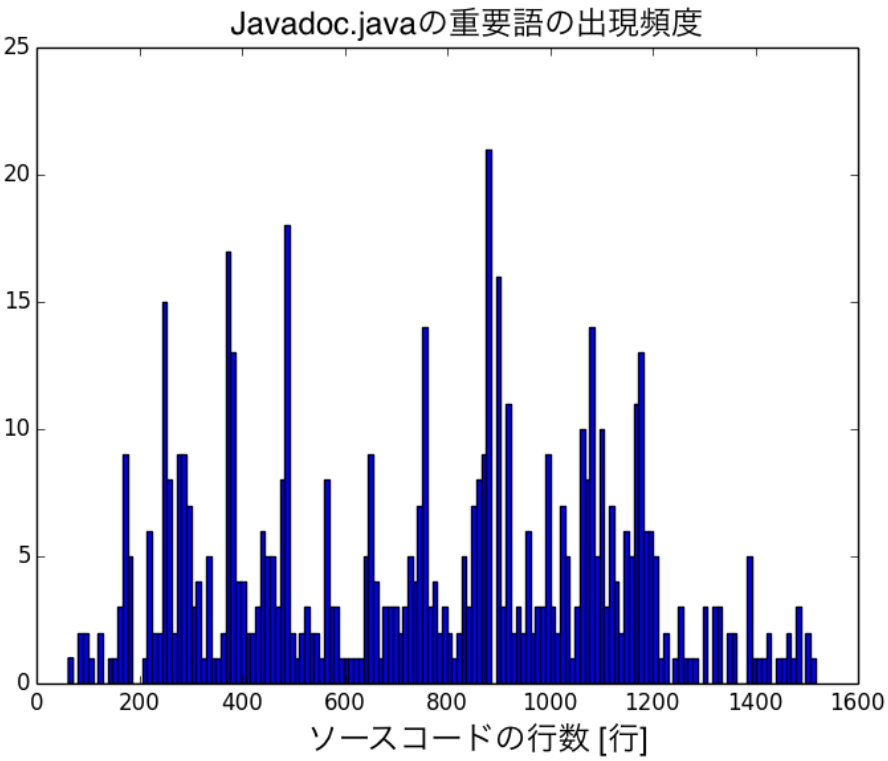

図 **5.2**　**2013** 年 **10** 月 **27** 日における **Javadoc.java** のヒストグラム

いる setter などの関数が重要部分だと特定できる．また，1050 行から 1200 行くらいは他と比べて平均的に値が高くなっている．これも一番値が高くなった 870 行から 940 行あたりと同様に「 toExecute.createArgument().setValue(」というコードが頻出していた．このコードを含む関数は 700 行から 800 行あたりにある関数 execute で実行されている．よって，コードの名前からも推測できるが，この関数 execute を実行するために何らか引数に値をセットする関数が重要部分だと考えられる．また，この関数 execute の 760 行あたりも値が 10 を超えていたり，少し高くなっていて，そのあたりで「 toExecute.createArgument().setValue(」というコードを含む関数群が呼びだされていた．

したがって,「Javadoc.java」において重要部分は大きく分けて 3 つ挙げられる．

- 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重要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重要部分として抽出することができた．

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

Apache Ant を用いた重要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重要語を用いて，10 行ごとの重要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重要部分抽出を行い 10 行ごとの重要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 5.3

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重要度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．図より，節 5.2 で説明した重要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重要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重要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重要部分抽出を行い 10 行ごとの重要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 5.4 である．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，節 5.2 説明した重要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重要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重要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，それに合わせて重要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，重要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重要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重要部分の位置の変化が少ないことがわかった．

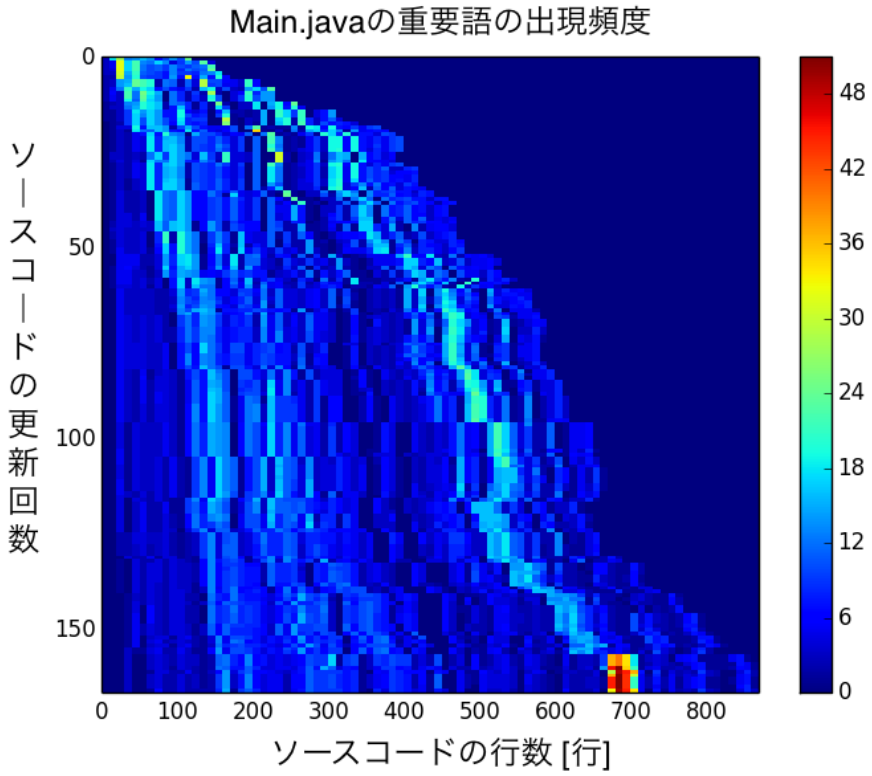

図 5.3　Main.java のヒートマップ

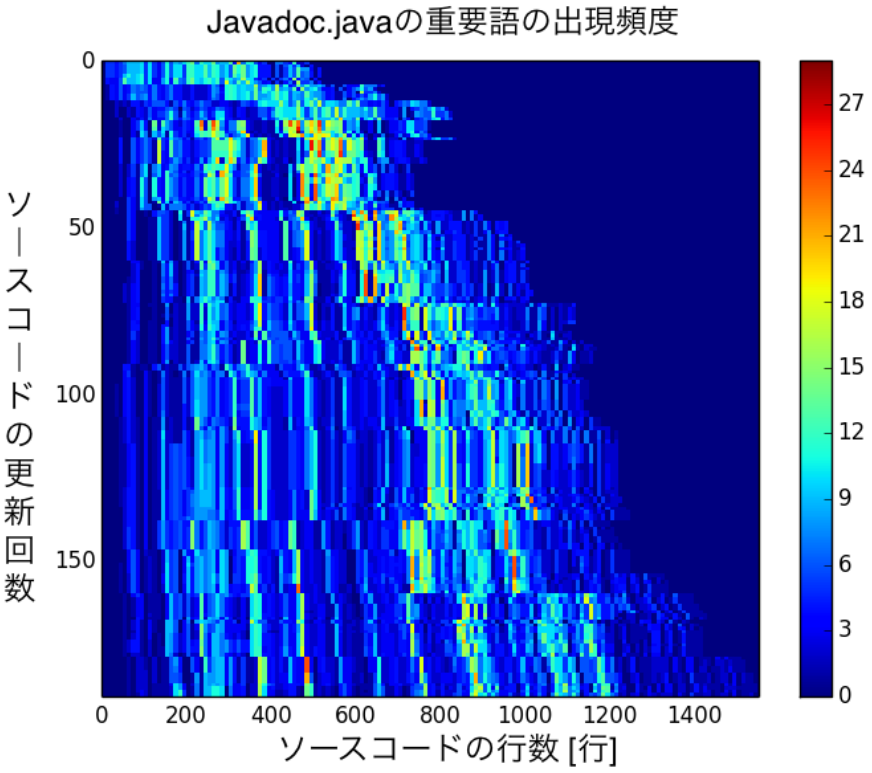

図 5.4　Javadoc.java のヒートマップ

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重要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重要語の抽出を行い，その重要語の行ごとの出現頻度を求めて重要部分の抽出を行った．

まず，ソースコード中から重要語を抽出できるのかという問いかけに対し，サンプルプログラムを用いて重要語抽出の評価実験を行い，ソースコードにおいても 3 語以上の単語 N-gram ならある程度の確率で重要語を抽出できることを示した．次に，ソースコード中から重要部分を抽出できるのかという問いかけに対し，Apache Ant の公開されているソースコードを用いて重要部分抽出を行い，ソースコードから重要部分の抽出をすることができることを示した．また，ソースコードの変更によって重要部分位はどのような変化があるのかという問いかけに対し，同じく Apache Ant の公開されているソースコードを用いて，ソースコードの変更による重要部分の変化について調べた．その結果，抽出した重要部分はソースコードの変更による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．

以上より，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，ソースコード中の重要語を抽出でき，その重要語の行ごとの出現頻度から重要部分を抽出することができた．また，その重要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．

また，今後の課題としてはサンプルプログラムを用いて重要語抽出を行った際に，単語 N-gram の重要語候補のヒット数が少なかった．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，それとは違うソースコードの特徴を利用した単語 N-gram の選択方法を考える必要があると考えた．

謝辞

本研究を行うにあたり，研究課題の設定や研究に対する姿勢，本報告書の作成に至るまで，全ての面で丁寧なご指導を頂きました，本学情報工学・人間科学系水野修教授に厚く御礼申し上げます．本報告書執筆にあたり貴重な助言を多数頂きました，ソフトウェア工学研究室の皆さんをはじめとする，学生生活を通じて著者の支えとなった家族や友人に深く感謝致します．

参考文献

[1] H.C Wu, R.W.P. Luk, K.F Wong, and K.L. Kwok, “Interpreting tf-idf term weights as making relevance decisions,” ACM Transactions on Information Systems, vol.26, no.3, pp.13:1–13:37, June 2008.

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

[3] K.S. Hasan and V. Ng, “Conundrums in unsupervised keyphrase extraction: Making sense of th state-of-the-art,” Coling, pp.365–373, Aug. 2010.

[4] J. Sivic and A. Zisserman, “Video google: A text retrieval approach to object matching in videos,” ICCV, pp.1470–1477, Oct. 2003.

[5] A. Aizawa, “An information-theoretic perspective of tf-idf measures,” Information Processing and Management, vol.39, pp.45–65, 2003.

[6] S. Robertson, “Understanding inverse document frequency: On theoretical arguments for idf,” Journal of Documentation, vol.60, no.5, pp.503–520, 2004.

[7] D. Hiemstra, “A probabilistic justification for using tfxidf term weighting in information retrieval,” International Journal on Dgital Libraries, vol.3, no.2, pp.131–139, Sept. 2000.

[8] K.S. Jones, “A statistical interpretation of term specificity and its application in retrieval.,” Journal of Documentation, vol.28, no.1, pp.11–21, 1972.

[9] D. Metzler, “Generalized inverse document frequency,” CIKM, pp.26–30, Oct. 2008.

[10] 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，” DEIM Forum，vol.A，pp.3–5，2015.

[11] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch, “Replacing suffix trees with enhanced suffix arrays,” Journal of Discrete Algorithms, vol.2, no.1, pp.53–86, March 2004.

[12] D. Okanohara and J. Tsujii, “Text categorization with all substring features,” SDM, pp.838–846, April 2009.

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

[14] T. Gagie, G. Navarro, and S.J. Puglisi, "New algorithms on wavelet trees and applications to information retrieval," Theoretical Computer Science, vol.426-427, no.6, pp.25–41, April 2012.