2020

自然言語処理
Java 版

1. はじめに

近年，ビッグデータやデータマイニングなど，大量にあるデータを処理することで，実社会における課題に対する解決のヒントを得たり，人間では分析しきれない特性などから，新たな課題を掘り起こしたりするなどの取り組みが行われている．対象とする一データとして，自然言語がある．自然言語とは，人間が言語コミュニケーションのために用いている言語を指す．日本人にとっては，日本語が最もなじみのある自然言語である．

自然言語をコンピュータが処理できるようにすることにより，人間では処理しきれない特徴を見出したり，人間が行ってきた言語コミュニケーションの一部をコンピュータが行ったりすることを可能にした．

本実験では，自然言語処理技術に関する，言語処理の基礎的な知識及び，開発環境を学習する．自然言語処理のプログラム開発を通して，自然言語に関する基礎的な分析方法の実際を把握し，利点や課題を知る．

2. 目標

自然言語処理技術を活用して，テキスト分析することを題材として，次を習得することを本実験の目標とする．

(1) 大量のファイルを読み込み，処理結果をファイルに出力できる
(2) 外部プログラムを使うことができ，その出力結果を利用できる
(3) 適したデータ処理の仕方や形式，データの保存形式を選び，利用できる

3. 予定

実験実習は全 5 回を予定している．初めの 3 回で，3 種類の分析手法を実装する．その後 2 回で，テキスト分析結果の質を高めるために，プログラムの改良を考え，レポートを作成する．

4. 概要

(1) 形態素解析器を利用して 3 種類の分析手法を実装する
(2) 3 種類の分析手法の意味と特徴をつかむ
(3) 分析結果の問題点を見つけ，改善する．

5. 実験環境

今回の実験では，表 1 の環境にて実験すること．

表 1　実験環境

項目	環境
Operating System	Windows（7 以上のバージョン）
統合開発環境	Eclipse
使用する言語	Java
レポート	TeX，Word など，環境を問わない．ただし，Word を使う場合は，TeX と同様のフォント，スタイル，文書構造，文書をとること. このことについて理解できない場合は, TeX を用いることを推奨する.

自然言語処理　第１回
（Term Frequency）

1. 形態素解析

テキストに含まれている単語を，文から逐次にたどっていくと，単語として成立する，すべての単語を抽出することができる．

例えば，「両生類」を検索語として検索した時，図 1 のように，文字列が一致する部分を検出し，該当する文書，および該当する語を含む文などを表示する．

> 両生類・爬虫類天然記念物一覧は、日本の文部科学大臣が指定する、天然記念物（特別天然記念物を含む、以下同）のうち、両生類・爬虫類に関わるもののリスト。天然記念物指定基準「動物」に基づき指定されたもののうち、両生類・爬虫類の種および生息地、繁殖地等を掲載する。なお、本項では文化財保護法に基づき国（日本国文部科学大臣）が指定した天然記念物を対象とし、地方自治体指定の天然記念物は対象外とする。

図 1 逐次検索のイメージ図

しかし，逐次にたどっていく方法では，その文が意図していない単語を抽出してしまう恐れがある．例えば，以下のような文があった場合には，どのように解釈されるだろうか．

> 技術者がこの先生きのこるためにすべきことはなにか

人が意味を理解して解釈するときには，

「技術者が，この先，生きのこるために，すべきことはなにか」

と，読点を付け加えられる状態で理解するのではないだろうか．しかし，逐次にたどっていくと

「技術者が，この先生，キノコるためにすべきことはなにか」

などと解釈されることがあり得る．

「先生」という単語に解釈されることの是非については別の観点からの検証が必要であるが，文書の意味から考えると，「先生」という単語として認識されないことが望まれる．そのように望まれる時，文法の規範や辞書をもとにして，形態素に分割するツールが形態素解析器である．

注意として，形態素解析器は，ある特定の文法や辞書に則って，特定の方法で抽出しているに過ぎないので，解析器によって異なる結果になったり，人間の感覚と合致しない結果になったりすることがある．

例えば，以下の文を形態素解析することを考えたとき，どのように解析されるであろうか．

> うらにわにはにわとりがいる

「裏庭には鶏がいる」，「裏庭に埴輪採りがいる」，「裏にワニ，埴輪採りがいる」など

が考えられる．形態素解析器が必ずしも，絶対的な分析結果を示すわけではないことには気を付けなければならない．

今回は，外部の形態素解析プログラムを呼び出して，その結果を利用して検索結果に重みを付ける方法を考える．単語の出現回数から与えられる重みで文書を特徴づける．

2. 形態素解析環境の整備

形態素解析器を利用したプログラムを作成する準備として，形態素解析器をインストールし，解析できることを確認する．

2.1 形態素解析ソフトウェアのインストール

形態素解析ソフトウェアとして MeCab を利用する．リンク（http://taku910.github.io/mecab/）から，MeCab のインストーラ（mecab-0.996.exe）をダウンロードし，ダブルクリックしてインストールする．文字コードの選択の際は，Windows では“Shift-JIS”とする．

(1) セットアップ開始

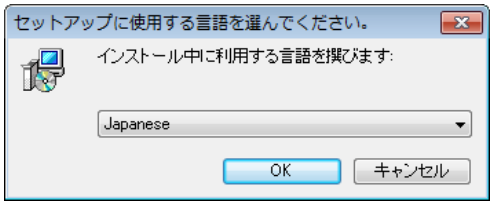


図 2 セットアップに使用する言語選択

(2) セットアップウィザードの開始

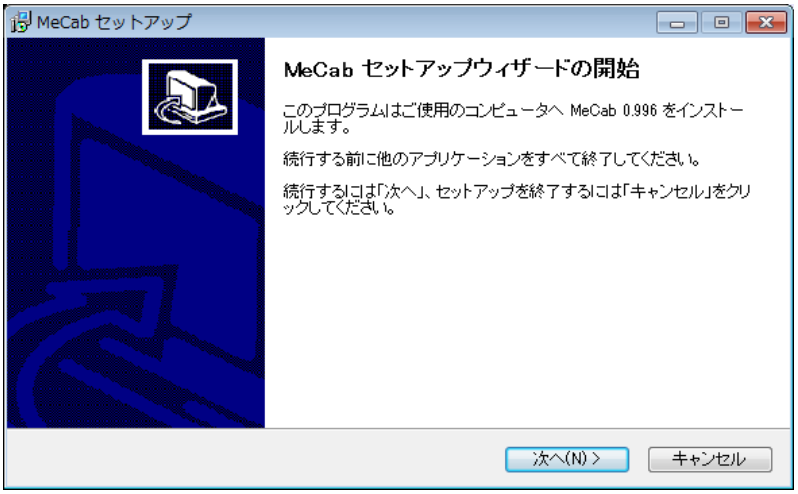


図 3　MeCab セットアップウィザード開始

(3) 文字コードの選択

文字コードとして「SHIFT-JIS」を選ぶ

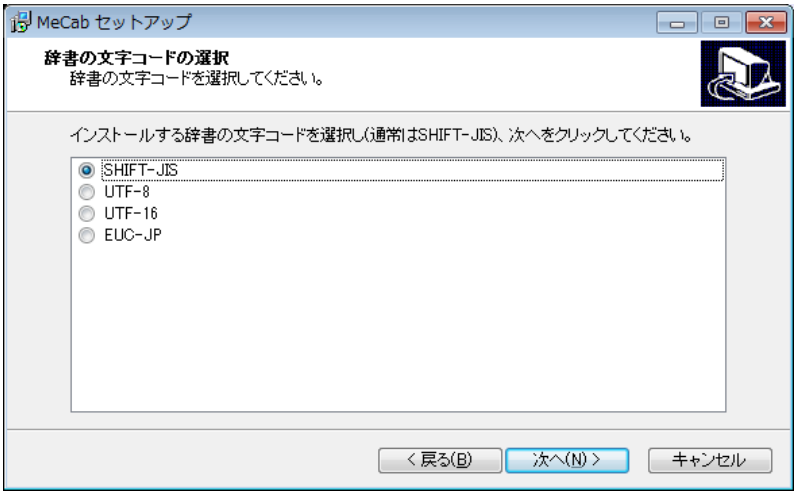


図 4　辞書の文字コードを選ぶ

(4) 使用許諾契約書の同意

使用許諾契約書を読み，同意する．

図 5　使用許諾契約書に同意する

(5) インストール先を確認する

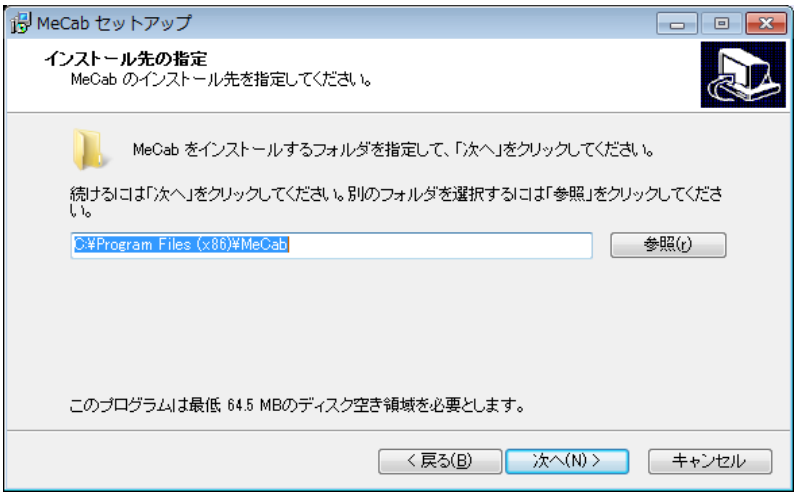


図 6　インストール先の選択

(6) ショートカットを作る

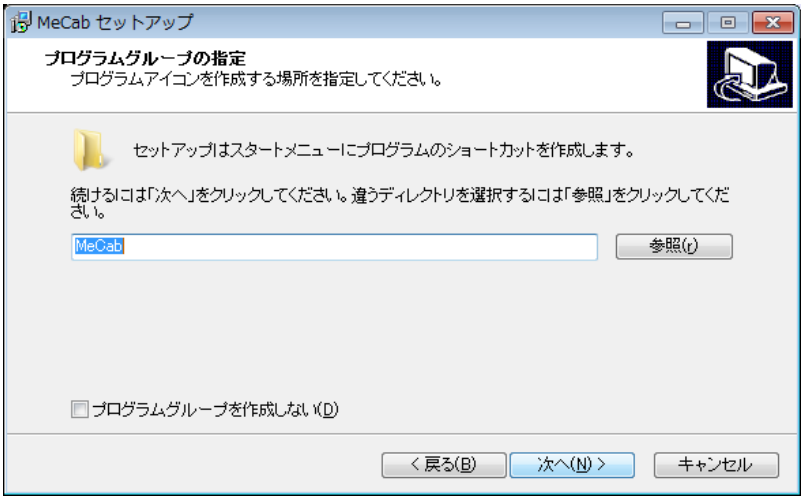


図 7　デスクトップにショートカットを作る

(7) インストール設定を確認する

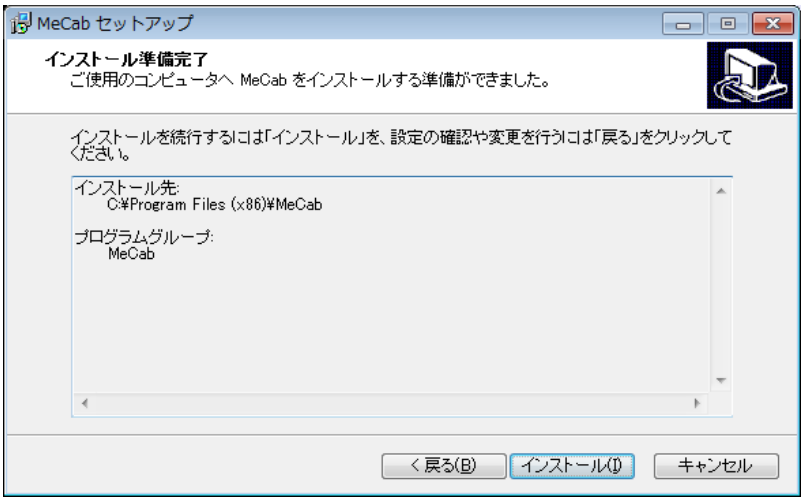


図 8　インストールを開始する

(8) セットアップの注意

他のユーザでログインしても MeCab を使えるようにする.

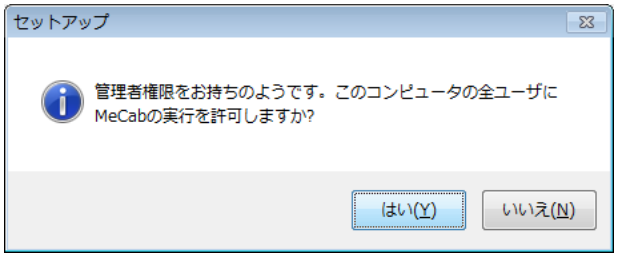


図 9　マルチユーザの実行許可

辞書の作成が開始する.

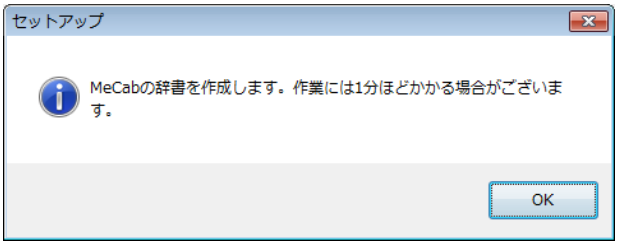


図 10　辞書作成の開始

セットアップが完了すると，図 11 のようなウィンドウが表示する.

図 12　セットアップの完了

(9) 実行

標準のままインストールすると，デスクトップに図 13 のような MeCab 実行ファイルへのショートカットができるので，起動して MeCab が動作することを確認する．図 14 に，「すすももももももものうち」を入力して，形態素解析の結果が表示されている様子を示す．

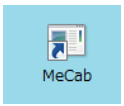


図 13　デスクトップアイコン

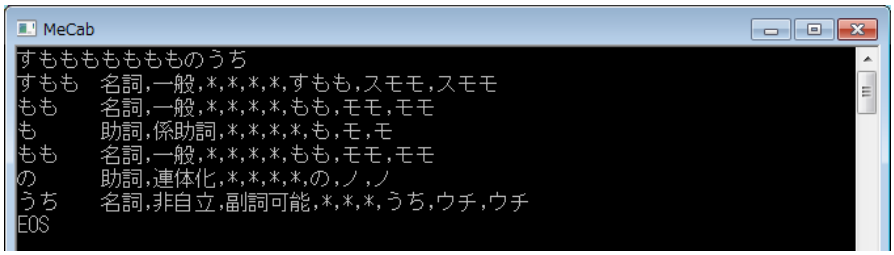


図 14　MeCab をショートカットアイコンから実行した様子

2.2 パスを通す

このままだと，インストールした実行ファイルを絶対パスで呼び出さなければ実行できないので，MeCab の実行ファイルと DLL ファイルへのパスを通すために，環境設定をする．

(1) Windows7

環境変数に含まれるシステム環境変数の Path として，以下を追加する．

［マイコンピュータ］のプロパティ

→［システムの詳細設定］

→タブ［詳細設定］
→［環境変数］の［システム環境変数］
→［変数］が「Path」になっている項目を選んで「編集」ボタンを押す.

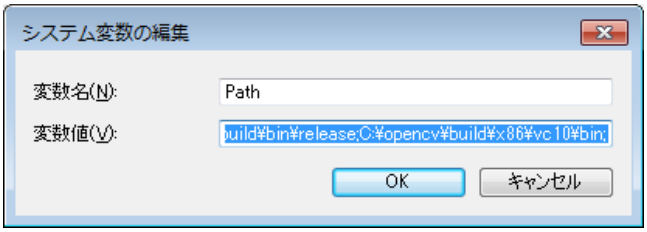


図 15　Path を設定するウィンドウ

既に変数値に入っている値の後ろに，セミコロン「；」の後に追加する.

```
C:¥Program Files¥Common Files¥Microsoft Shared¥Windows Live;%SystemRoot%¥system32;%SystemRoot%;%SystemRoot%¥System32¥Wbem;%SYSTEMROOT%¥System32¥WindowsPowerShell¥v1.0¥;C:¥Program Files¥Windows Live¥Shared;C:¥Program Files¥Intel¥WiFi¥bin¥;C:¥Program Files¥Common Files¥Intel¥WirelessCommon¥;c:¥Program Files¥Common Files¥Ulead Systems¥MPEG;c:¥Program Files¥Microsoft SQL Server¥100¥Tools¥Binn¥;c:¥Program Files¥Microsoft SQL Server¥100¥DTS¥Binn¥;C:¥Program Files(x86)¥MeCab¥bin;
```

リスト 1　新しく MeCab の設定を追加した後の Path 設定の例

```
C:¥Program Files(x86)¥MeCab¥bin;          //64bit OS の場合
C:¥Program Files¥MeCab¥bin;               //32bit OS の場合
```

リスト 2　追加するパス

追加するときには，既存のパスの後ろに追加する．追加するパスの文字列は，テキストを写したり，手打ちをしたりしない．実際に格納されているフォルダのパスをコピーすること.

設定後，図 16 のように，コマンドプロンプトから MeCab を呼び出して，解析したい文字列を入力したときに，出力として，形態素解析された結果が表示されることを確認する．デスクトップのショートカットからではなく，コマンドプロンプトから実行できることを確認する．これで，mecab.exe がある場所へのパスが通ったということになる.

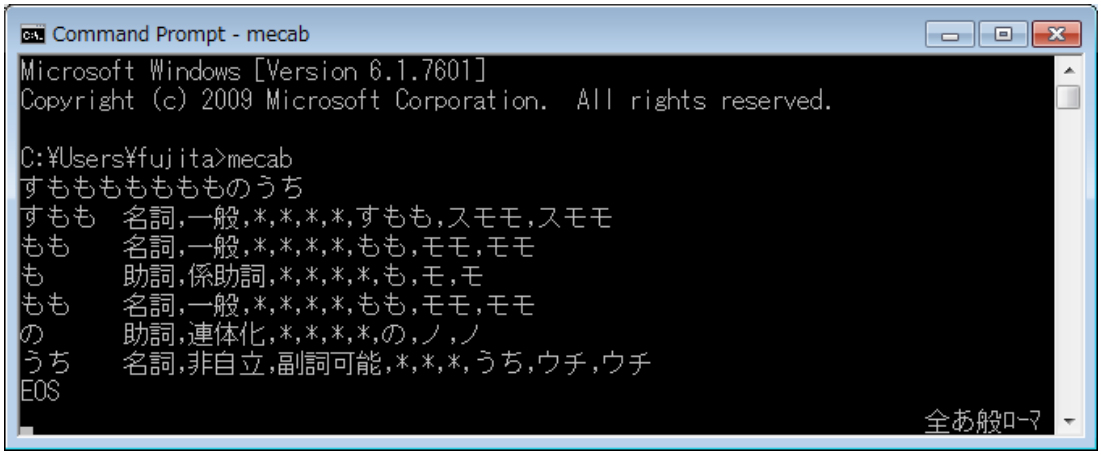


図 16　MeCab がよびだせることを確認する

(2) Windows10

Windows10 の場合の Path 設定方法を示す.

(a) PC の「プロパティ」を開く

デスクトップなどの PC アイコンから「プロパティ」を開くと，図 17 のようなウィンドウが表示される．左側のメニューから「システムの詳細設定」を開く．

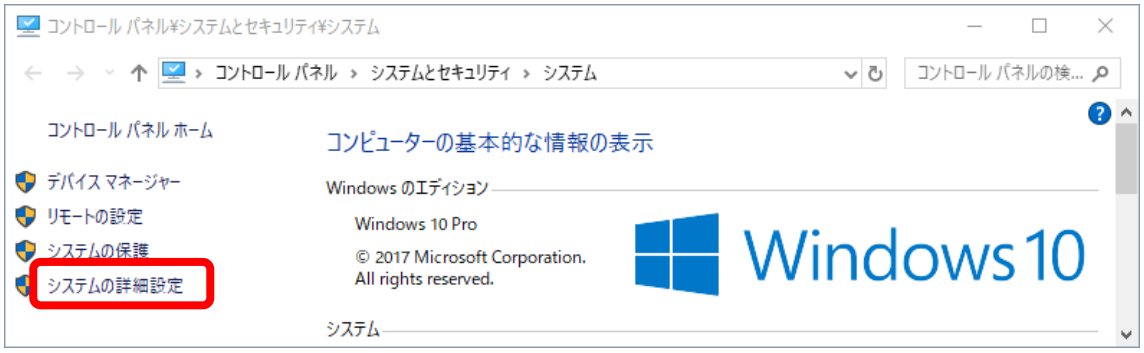


図 18　PC の「プロパティ」

(b) **「システムのプロパティ」の「詳細設定」から「環境設定」を押す**

図 19 のように，「詳細設定」のタブを開き，「環境変数」ボタンを押す.

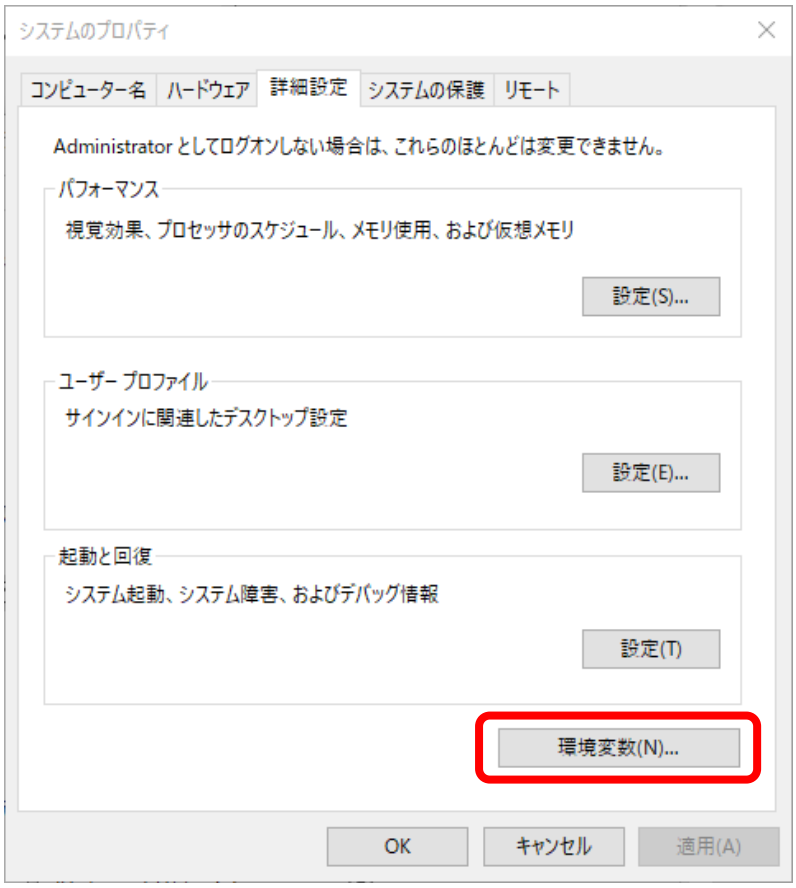


図 20　「システムのプロパティ」

(c) **「システム環境変数」の「Path」を編集する**

図 21 のように,「システム環境変数」の「Path」を選び,「編集」を押す.

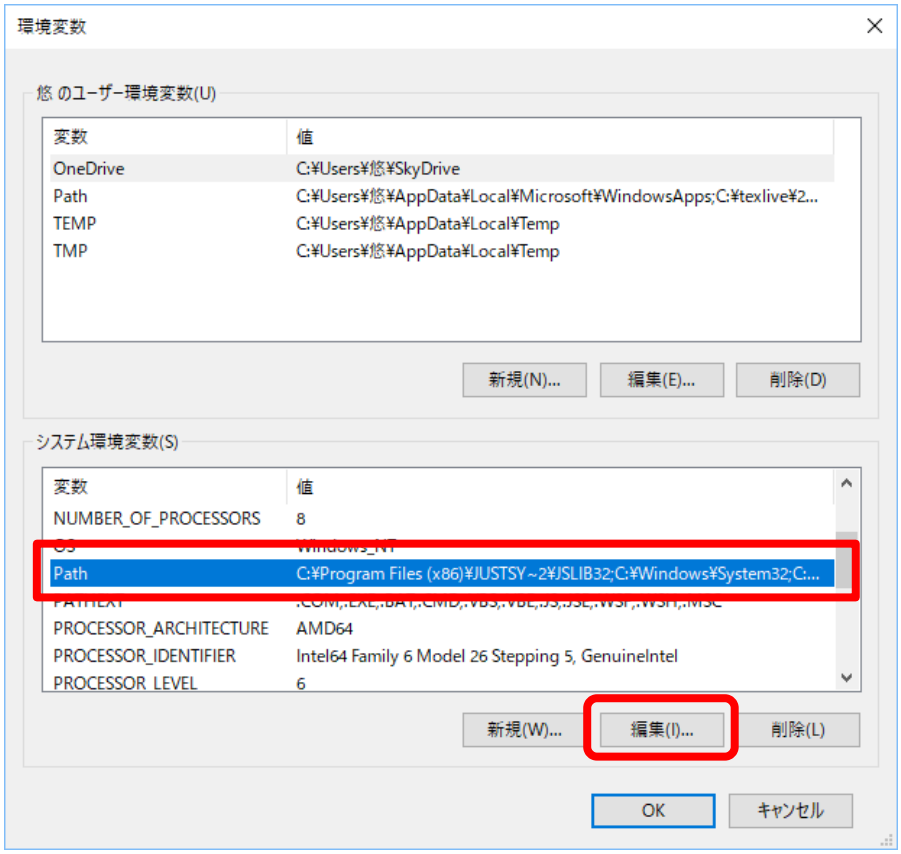


図 22　システム環境変数の Path

(d) 編集画面を開く

図 23 のように，現状登録されている Path の一覧が表示される．

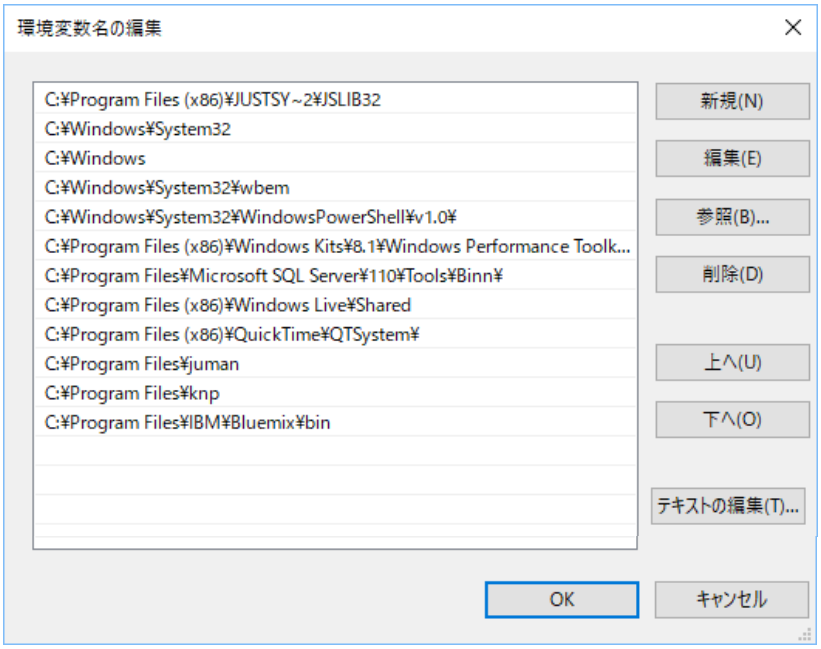


図 24　Path 編集画面

(e) 追加する Path をインストール先のディレクトリからコピーする

追加したい Path を用意するために，図 25 のように，追加したいフォルダを開き，その Path をコピーする．

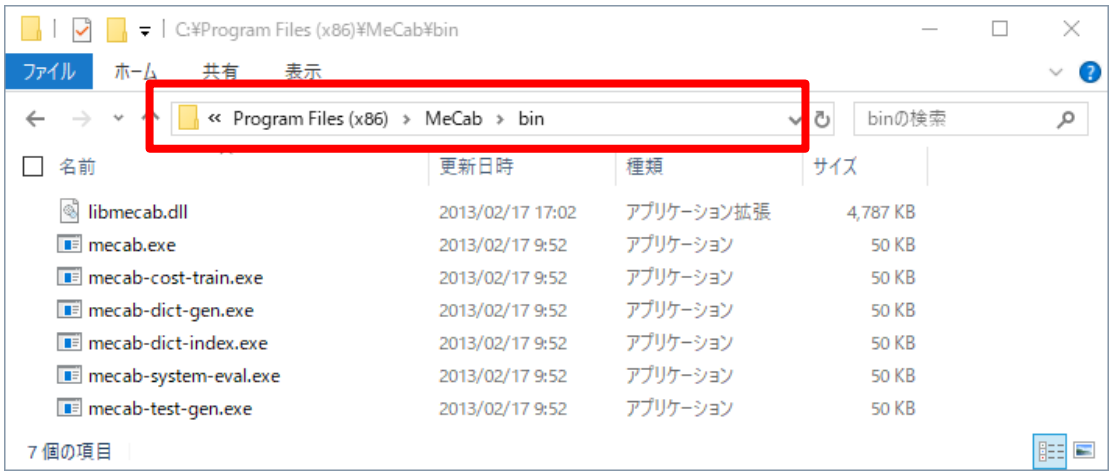


図 26　追加する Path のコピー

(f) 「新規」から追加する

図 27 のように，編集画面にて「新規」ボタンを押す．

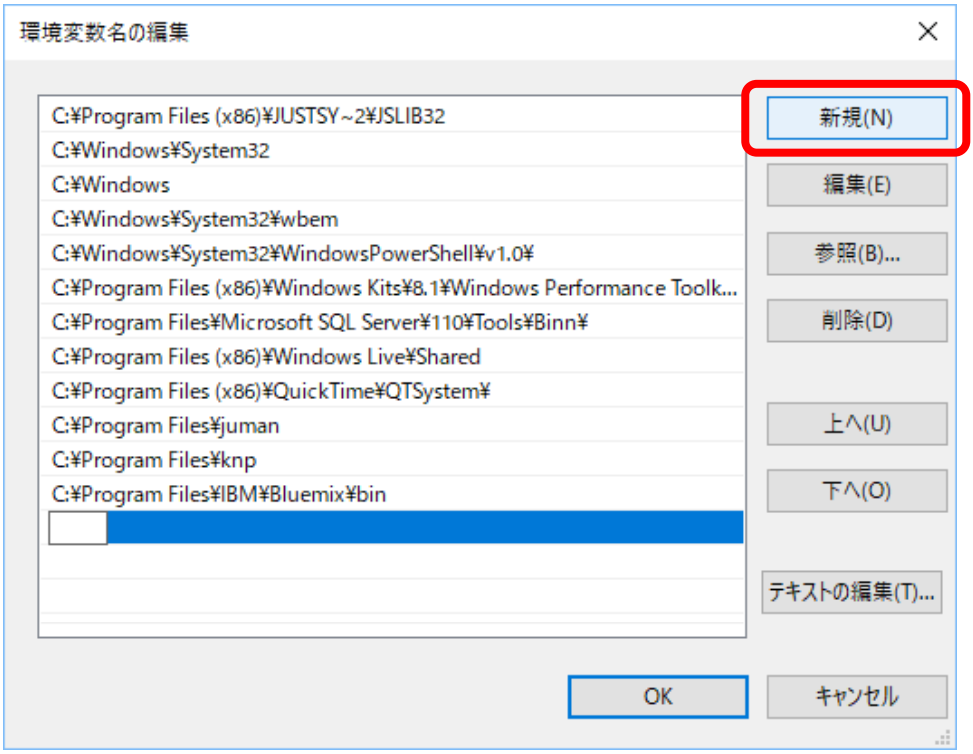


図 28　新規に Path を追加する

(g) 追加後の様子

新たに Path を追加すると, 図 29 のように, 追加されていることが確認できる.

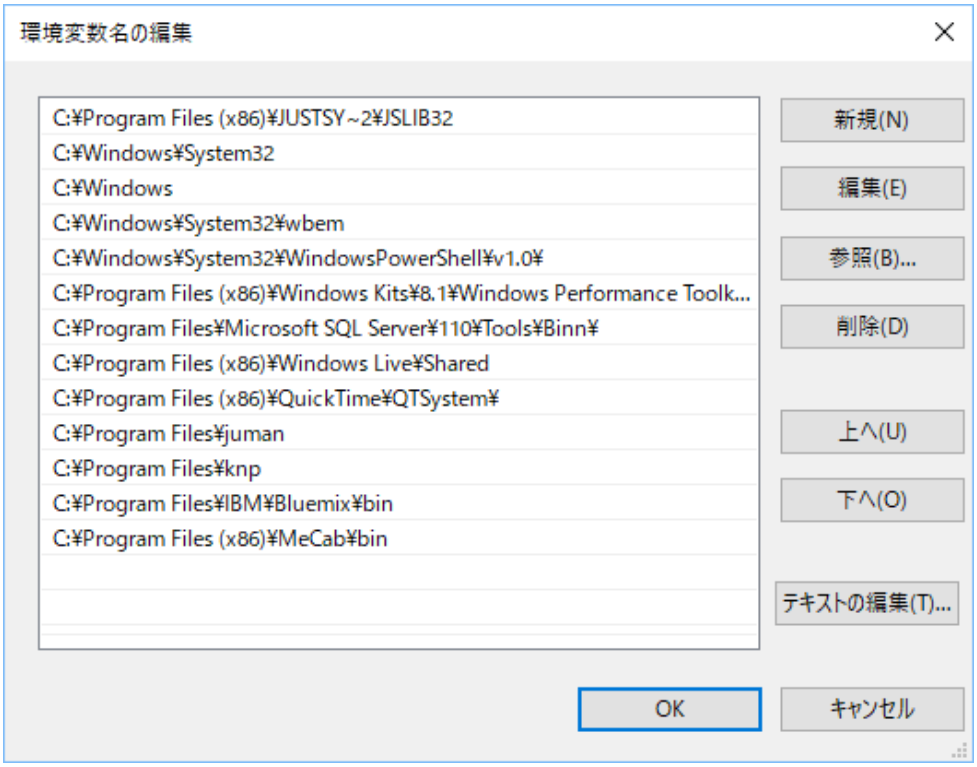


図 30　新規 Path 追加後

3. プロジェクト

(1) プロジェクト作成

「NaturalLanguageProcessing」プロジェクトを作り，その中に，「nlp」パッケージを作り，その中にクラスを作成していくこととする．

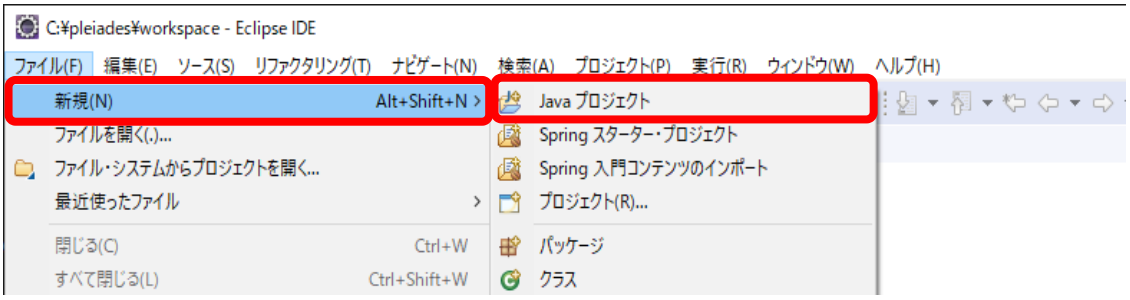


図 31　プロジェクト作成を始める

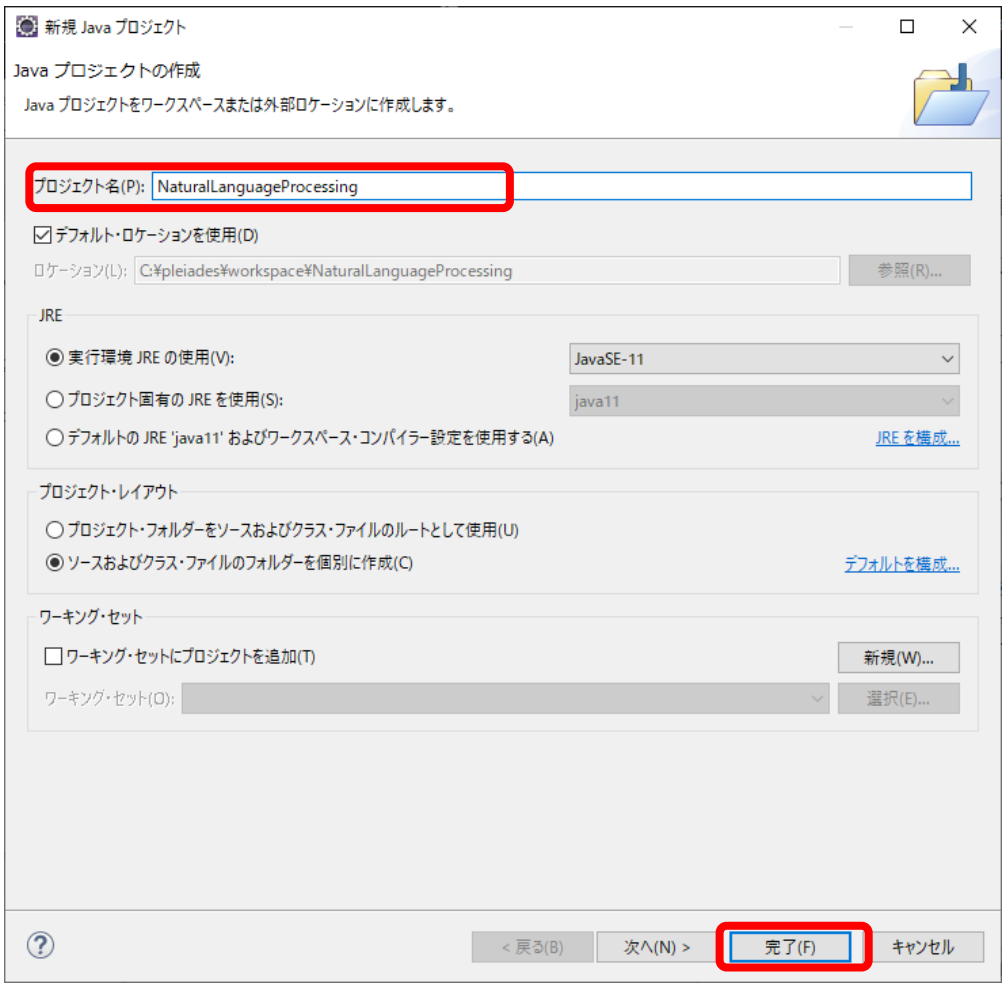


図 32　プロジェクトの設定

(2) パッケージ作成

対象とするプロジェクトにて「src」ディレクトリにて右クリックして「新規／パッケージ」にて新規にパッケージを追加する．

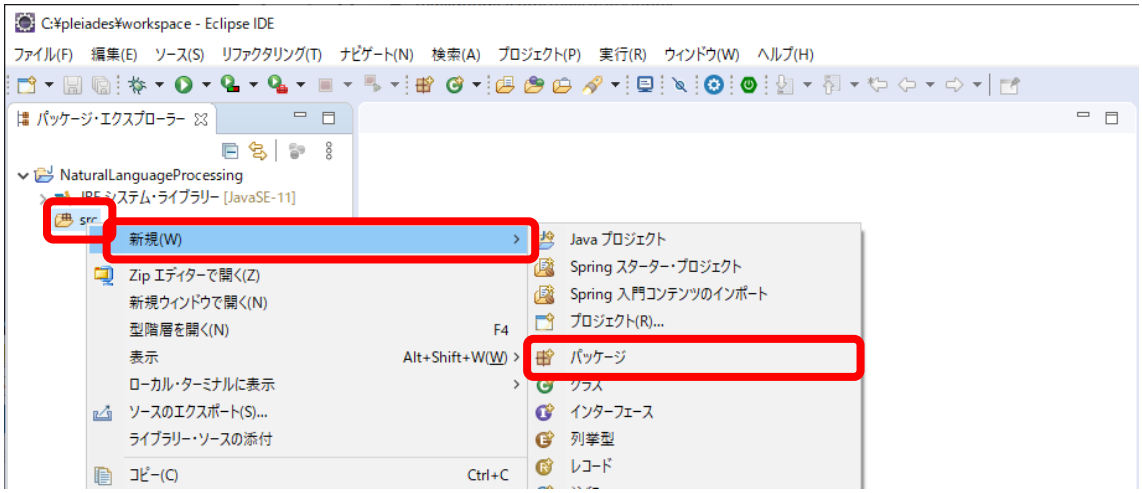


図 33　パッケージ作成を始める

パッケージ名を「名前」欄に記入する．完了を押してパッケージを作る．

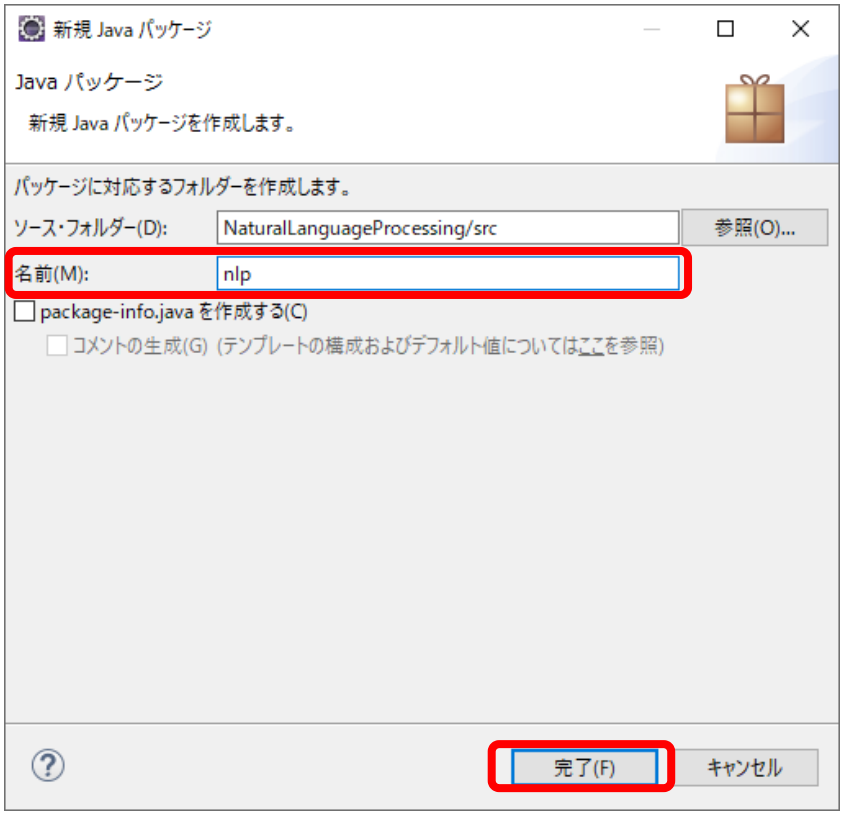


図 34　パッケージ設定

4. *tf* (term frequency)

4.1 概要

tf とは,「語の出現頻度」に着目した重み付けである．たとえば，あるウェブページに「自動車」という語が 100 回出現していて，別のページには 20 回出現している場合，100 回出現するページの方が自動車という語句にふさわしいページであるとみなす考え方である．

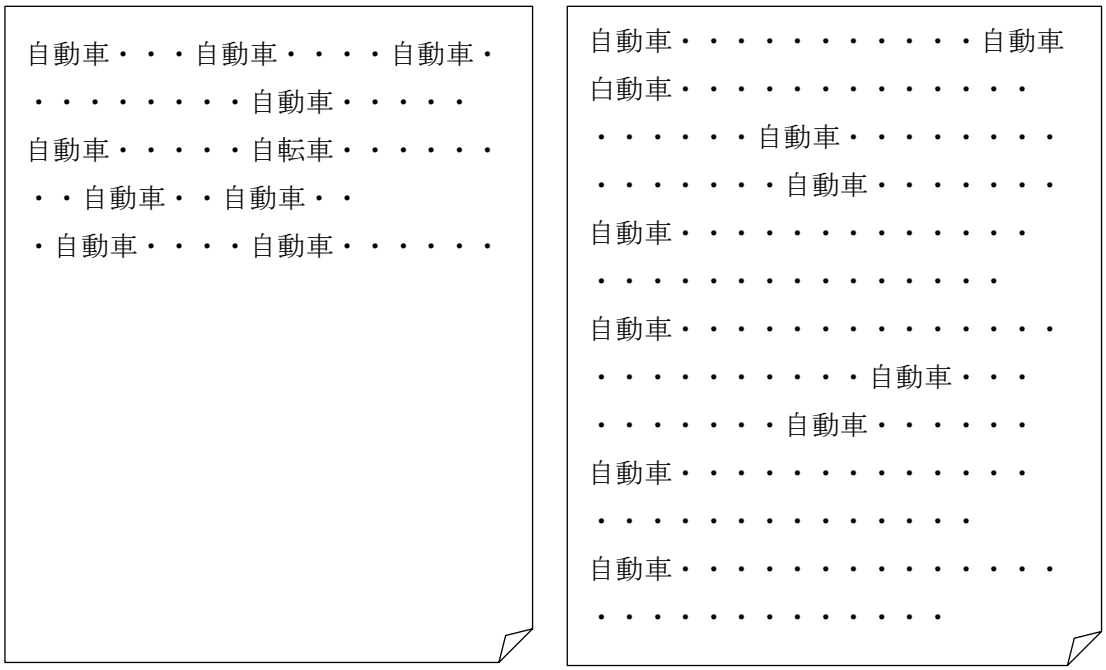
tf を算出するために，形態素解析ソフトを利用することによって，たとえば，手元にある 100 個のテキストファイルに含まれるテキストを自動的に品詞ごとに分割することが可能となる．その中に出現する語を計数することによって *tf* を算出し，重み付けを行う．

このようにして付与した重みを利用して，ユーザが入力した キーワードに合致する重みが大きい順にテキストを並べると，テキストの特徴を表す量を数値化できる．

4.2 正規化

単語の出現回数はテキストの量に依存するので，取得してきた出現頻度をそのまま利用するのは好ましくない. たとえば, 自動車という語が, 200 文字のテキスト A に 100 回出現することと，20000 文字のテキスト B に 100 回出現することを比較して，テキスト A と B では同一頻度であるといった判定はふさわしくない．

具体例を挙げると，図 35 のように，共に「自動車」が 10 回出現する文章だが，5 行の文章と 13 行の違いがある．このとき，どちらも出現頻度が同じであるといえるだろうか．明らかに，5 行に 10 回出現する文章の方が，出現頻度が高いといえる．このような，違いを吸収させるような処理を施す．



5 行の文章に 10 回　　13 行の文章に 10 回

図 35　どちらの文書のほうが「自動車」の出現頻度が高いだろうか

出現回数を総単語数で割った値，すなわち重みを全て合計すると 1 になるように正規化する．文書 d における，単語 t の tf による重み$w_{tf_t^d}$は式(1)で定義される．

$$w_{tf_t^d} = \frac{tf(t,d)}{\sum_{s \in d} tf(s,d)} \tag{1}$$

分子の$tf(t,d)$は単語 t は文書 d 内に出現する回数，分母$\sum_{s \in d} tf(s,d)$は文書 d 内に出現する全ての単語の数である．

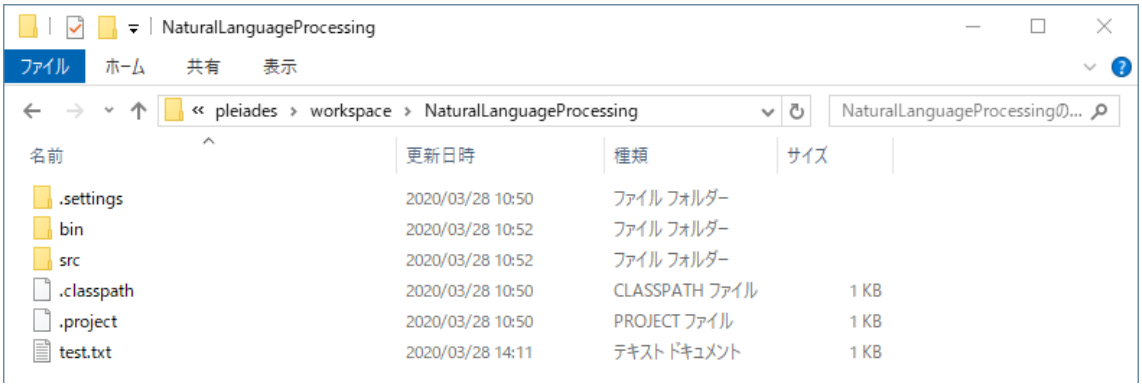
5. 実験

5.1 Mecab でファイルを解析する

MeCab でテキストファイルの内容を形態素解析して，その結果をプログラムで受け取る処理を確認する．

(1) 試行用ファイルの準備

試行するテキストファイル「text.txt」をプロジェクトフォルダに設置する．Java で作成したプログラムを実行するときの相対パスの基準となる場所は，このプロジェクトフォルダになる．



(2) クラスを定義する

TF の処理を「TermFrequency」クラスに実装することにする. 追加先のパッケージ「nlp」を右クリックして「新規／クラス」を選ぶ.

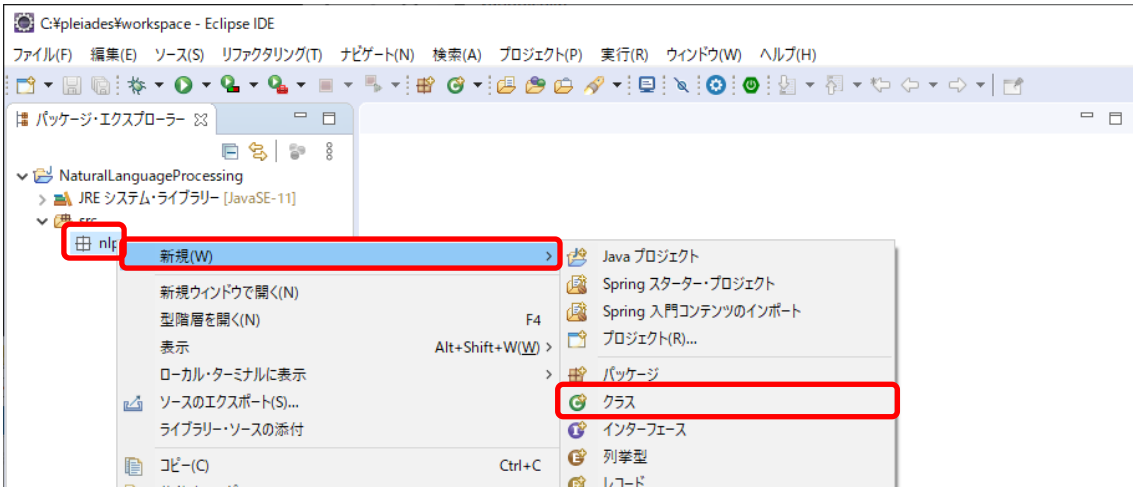


図 36　新規クラスの作成

新規クラスを設定するためのウィンドウが表示されるので，クラス名を「TermFrequency」とする. ここで main 関数を自動生成するように設定すると楽になる.

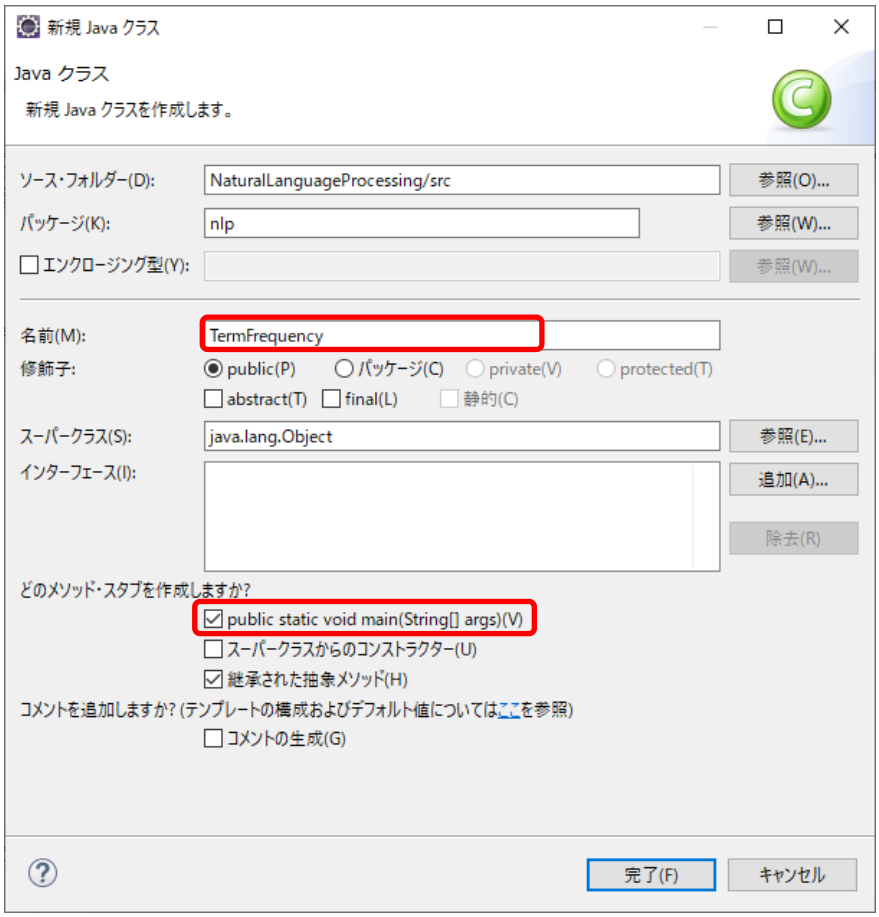


図 37　新規クラスの設定

(3) サンプルプログラム

試行用に設置した「test.txt」を MeCab で解析して表示するためのサンプルプログラムをリスト 3 に示す.

リスト 3　MeCab の試行用プログラム

```
package nlp;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class TermFrequency {

    public static void main(String[] args)
    {
        //解析対象のファイル名
        String inputFilename = "test.txt";
        //mecab コマンドで対象ファイルを解析するコマンド文
        String[] command = { "cmd.exe", "/C", "mecab", inputFilename };

        try
        {
            //解析するコマンドを実行する
            Process ps = Runtime.getRuntime().exec(command);
            //解析した結果を表示するためのオブジェクトに変換する
            BufferedReader bReader_i
              = new BufferedReader(new InputStreamReader(ps.getInputStream(), "SJIS"));
```

```
            // 標準出力を 1 行ずつ受け取る一時オブジェクト
            String targetLine;
            while (true)
            {
                // 形態素解析結果を 1 行ずつ受け取る
                targetLine = bReader_i.readLine();
                if (targetLine == null)
                {
                    //最終行になったら終わる
                    break;
                }
                else if (targetLine.equals("EOS"))
                {
                    continue;
                }
                else
                {
                    //行ごとに結果を表示する
                    System.out.println(targetLine);
                }
            }
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

この結果，テキストファイルに記載された内容が形態素解析されて表示されることを確認する．

5.2 tf を算出する

はじめに，100 件中 1 件のファイル（001.txt）を開き，そのファイルを読み込み，形態素解析して，単語をカウントして，tf による重みづけしたファイルを出力するところまでを実現する．出力ファイルの例として，リスト 4 のようになる．

（単語）	（出現回数）	（tf 値）
天然記念物	16	0.071111
県	11	0.048889
両生類	7	0.031111

リスト 4 term frequency のデータを出力する形式の例

(1) データファイルを設置する

データファイルの置き場所については，ソリューションフォルダの中に data フォルダに格納した位置に置くことにする．この場所は，プロジェクトフォルダと同じ階層の場所であるので，プロジェクトフォルダ内からは，1 段階上位の場所にある．

この場所は，プログラムを作成するときに，データファイルの場所を指定するときに重要になってくるので，この場所に位置させていることを把握しておく．

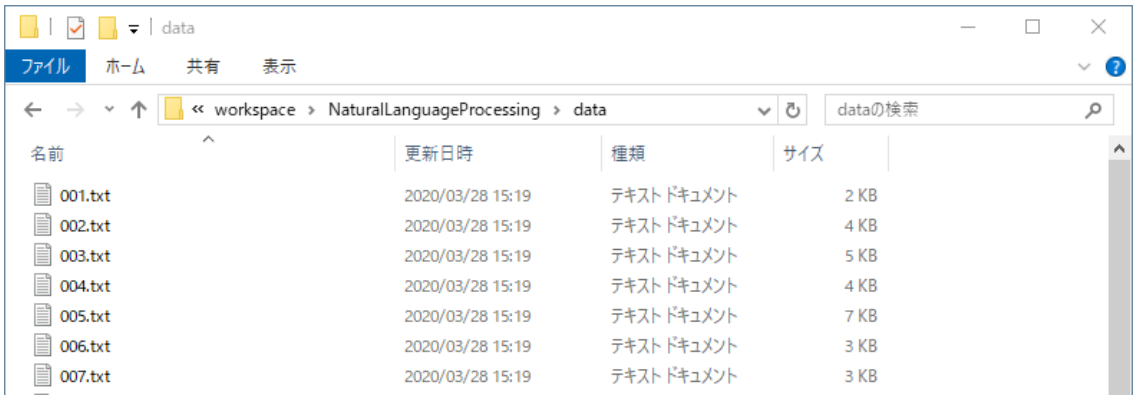


図 38　data フォルダの位置

(2) 1 件のファイルを解析する

まずは，「001.txt」というファイル 1 件を解析して，その結果を解析する．

```
String inputFilename = "data¥¥001.txt";
```

素性情報は，辞書によって異なるが，今回用いている辞書である ipadic の場合には，9 種類の素性が含まれている．例として，「渡し」という単語を解析したときに表示される素性情報（図 39）について，9 種類の素性を整理すると，図 40 のようになる．

```
渡し　　動詞,自立,*,*,五段・サ行,連用形,渡す,ワタシ,ワタシ
```

図 39　「渡し」を解析した結果の出力

素性情報	「渡し」の場合の値
品詞	動詞
品詞細分類 1	自立
品詞細分類 2	*
品詞細分類 3	*
活用型	五段・サ行
活用形	連用形
基本形	渡す
読み	ワタシ
発音	ワタシ

図 40　「渡し」の素性情報

そこで，解析結果を「¥t」または「,」にて分割して，分割された情報を格納する．

```
String targetArray[] = targetLine.split("[¥t|,]");
```

この結果，配列に，形態素解析した結果が格納される．

配列	素性情報	「渡し」の場合の値
targetArray[0]	表層形	渡し
targetArray[1]	品詞	動詞
targetArray[2]	品詞細分類 1	自立
targetArray[3]	品詞細分類 2	*
targetArray[4]	品詞細分類 3	*

targetArray[5]	活用型	五段・サ行
targetArray[6]	活用形	連用形
targetArray[7]	基本形	渡す
targetArray[8]	読み	ワタシ
targetArray[9]	発音	ワタシ

(3) 解析結果をオブジェクト化する

分割された情報を格納するためのクラスを用意する．

```
class Word
{
    private String hyousoukei = null;//表層形
    private String hinshi = null;//品詞
    private String hinshi1 = null;//品詞細分類 1
    private String hinshi2 = null;//品詞細分類 2
    private String hinshi3 = null;//品詞細分類 3
    private String katsuyoKata = null;//活用型
    private String katsuyoKei = null;//活用形
    private String genkei = null;//原形
    private String yomi = null;//読み
    private String hatsuon = null;//発音
}
```

さらに，これらのセッター，ゲッターを設置する．これは自動で生成させよう．セッターとゲッターを生成させたいフィールドを選んで，右クリックして「ソース／getter および setter の生成」を選ぶ（図 41）．

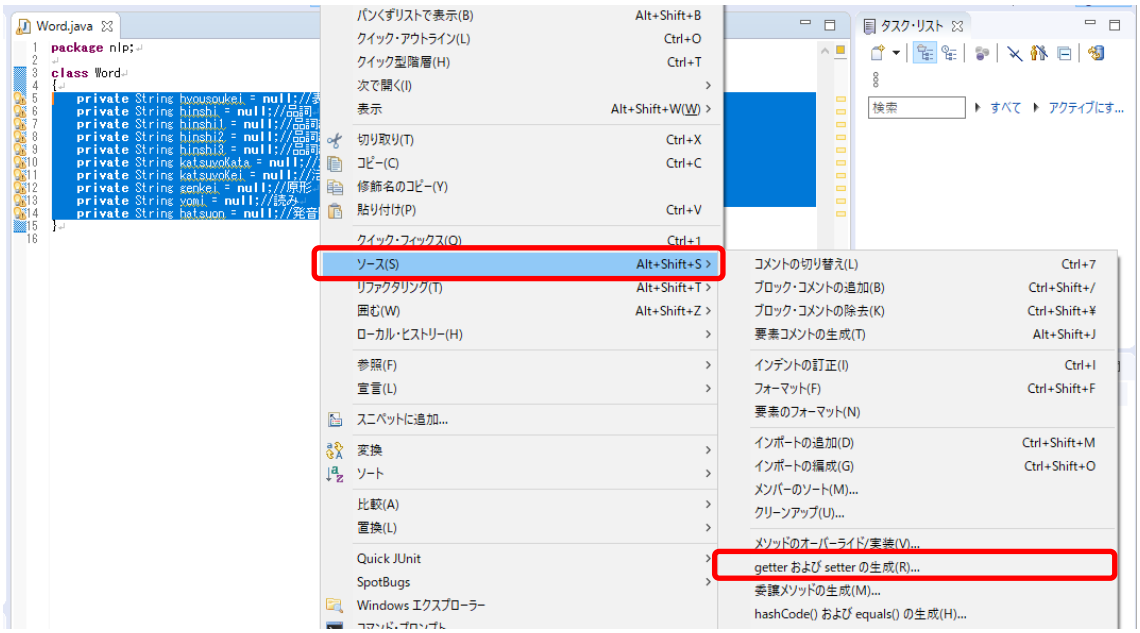


図 41　セッターとゲッターを生成する対象を選んで始める

生成する場所と対象のフィールドを選択する．「挿入ポイント」にて，挿入する場所を選ぶ（図 42）．

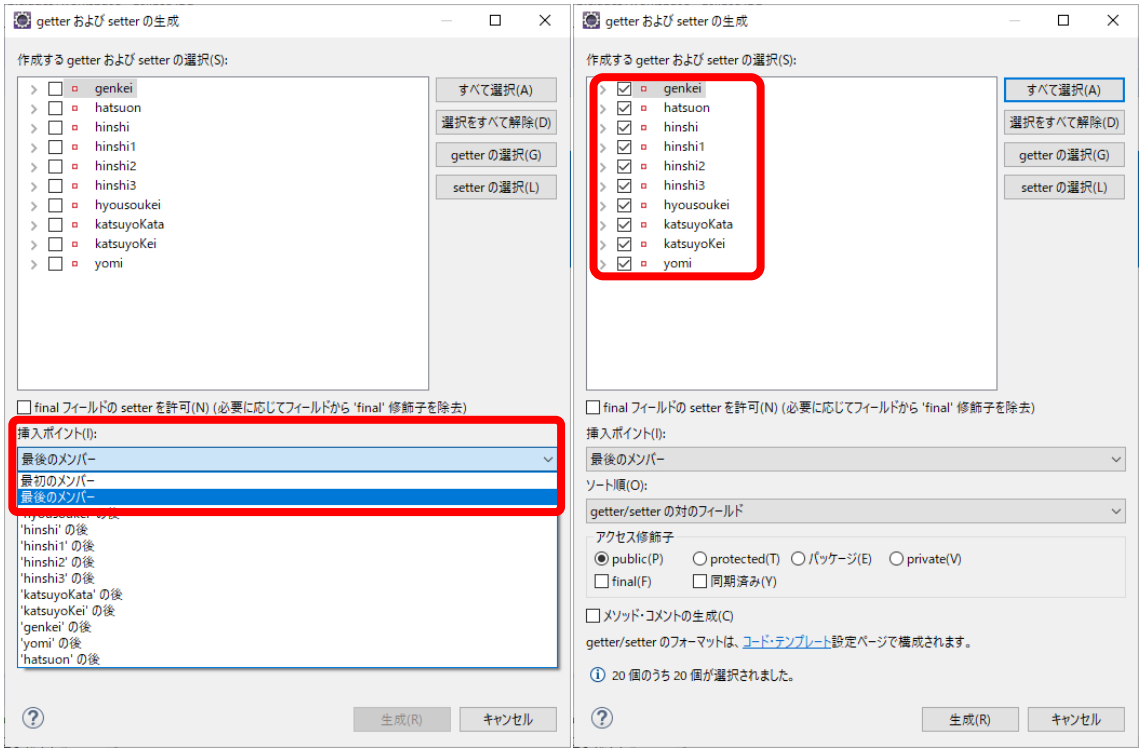


図 42　setter と getter の挿入場所と対象フィールド

挿入が完了すると，以下のようなセッターとゲッターが挿入される．

```
    public String getHyousoukei()
    {
        return hyousoukei;
    }
    public void setHyousoukei(String hyousoukei)
    {
        this.hyousoukei = hyousoukei;
    }
```

配列に分けられた形態素解析結果をオブジェクトに格納する．Word クラスをインスタンス化して，そのフィールドに値を設定していく．

```
Word wo = new Word();
if (targetArray.length >= 1)
    wo.setHyousoukei(targetArray[0]);
if (targetArray.length >= 2)
    wo.setHinshi(targetArray[1]);

 (続きは各自で入力してみよう)
```

(4) 単語の数を数える

分かち書きされた結果から，それぞれの単語の出現回数を数える．数える方法としては，分かち書きした単語をリストアップし，カウントアップする方法をとることにする．リストアップする方法として，リスト構造をとるデータ形式を利用する．

リスト構造を選択した理由として，何種類の単語が現われるのか，わからないので，新しい単語が現われるたびに追加できるデータ構造をとらなければならないことがあげられる．後から要素を追加できる構造をとる方法として，2 種類の方法を示

す．それぞれの方法のメリット，デメリットを考える．他にも方法があれば，それを用いてもよいが，いずれにしても，適した方法を用いる．

(a) 配列を使う

単語の文字列と，単語の出現個数からなる構造体による配列を構成し，新しい単語が現われたら，追加する．

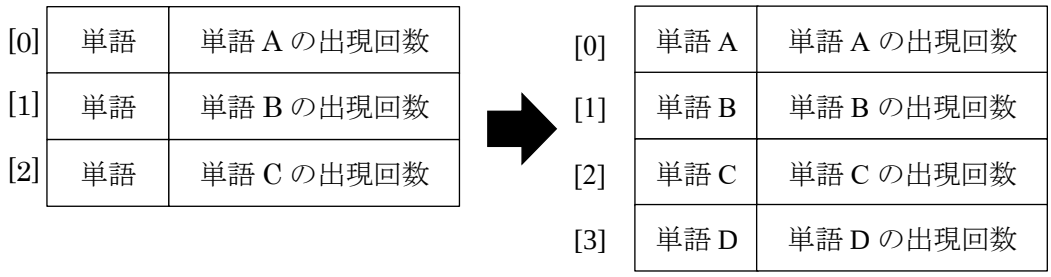


図 43　データ構造に配列を用いた場合の例

配列を使うときには，新しい要素が追加されたときには，配列の大きさを変更うることもできるが，巨大な配列を作るための，連続した領域を確保することができない場合には，エラーになってしまうことがある．したがって，データ件数が多くなることが予想される場合には，避けた方がよい．

(b) リストを使う

単語の文字列と，単語の出現個数と，次のアドレスからなる構造体を連結したリストをつなげる．この方法では，新しい単語が検出されたときに，その単語用の構造体を確保して，それまでにリストアップされた単語の構造体の最後に追加する．

例えば，単語 A，単語 B，単語 C がリストアップされていて，新たに単語 D が検出されたときに，構造体を追加して連結するイメージを図 44 に示す．

リスト構造をとることで，連続したメモリ領域を確保する必要がないので，メモリ資源を使える範囲で，大量のデータ領域を活用することができる．

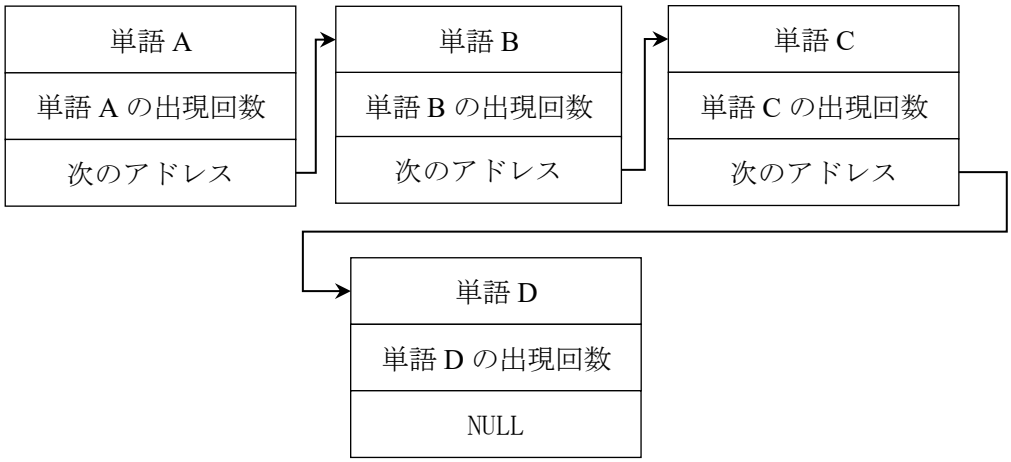


図 44　データ構造にリストを用いた例

Java では，多く用いられるデータ型については，用意されているので，その中に使えるデータ型がないか見てみる．ここでは，ArrayList を用いてみよう．

ArrayList は，格納するデータの型を指定する必要がある．解析した語と，それをカウントした数値，さらに，カウントした数値から導出される TF 値を格納することにする．これらのクラスを構成してみよう．

```
class WordCount
{
    private Word word;//形態素解析した語の結果を格納する
    private Integer count;//出現回数を計数するためのカウンタ

    WordCount(Word word,Integer count)
    {
        this.word = word;
        this.count = count;
    }
}
```

```
class TfCount extends WordCount
{
    Double tf;//計数値を全語数で割った TF 値
    TfCount(Word word, Integer count)
    {
        super(word, count);

    }

    TfCount(Word word, Integer count, Double tf)
    {
        super(word, count);
        this.tf = tf;

    }
}
```

このクラスについても，getter，setter を生成しておく．このクラスを用いた，リストを構成する．

```
ArrayList<TfCount> list = new ArrayList<TfCount>();
```

このリストを用いて，語の出現個数を計数する．

形態素解析した語ごとに，リストを先頭から順にたどっていき，見つけたい単語が見

つかったら，その語の出現回数をインクリメントする．リストに語がなければ，リストにその語を新たに追加する．

```
//リストの中に同じ語のエントリがあるか調べて，既にエントリがあれば，カウントアップする
int i;
for (i = 0; i < list.size(); i++)
{
   if (list.get(i).getWord().equals(wo))
   {
      list.get(i).setCount(list.get(i).getCount() +1);
      break;
   }
}
//リストにエントリが無かったときは，新しい語としてリストに追加する
if (i == list.size())
{
   list.add(new TfCount(wo, Integer.valueOf(1)));
}
```

ここで，形態素解析した語と，リスト内の語が一致するかどうかを検査する部分に注目する．形態素解析した結果は Word クラスのオブジェクトとして格納されている．このオブジェクトが一致しているかどうかを検査することになるが，「==」という等号を用いた検査を用いていない．これはなぜだろう．

等号を用いると，オブジェクトとして等しいかどうかを検査することになる．オブジェクトに含まれる内容がすべて一致していても，別のオブジェクトであるとみなされてしまうのである．

今回見たいのは，同じ形態素解析結果であるかどうかであるので，内容が一致していることを見たいので，それを検査する方法を用意する必要がある．それを用意する方法は，equals メソッドを定義することである．

リスト 5　新しい単語をリストに追加する

```
class Word
{
    （略）
   public boolean equals(Object obj)
   {
      //オブジェクトが null の場合，不一致
      if(obj == null)
      {
         return false;
      }

      //オブジェクトの型が異なる場合，不一致
      if(!(obj instanceof Word))
      {
         return false;
      }

      //オブジェクトの全てのフィールドが一致している場合，一致
      if(
        ((((Word)obj).getHyousoukei() == null && this.getHyousoukei() == null)
            ||((Word)obj).getHyousoukei().equals(this.getHyousoukei()))
      && ( (((Word)obj).getHinshi() == null && this.getHinshi() == null)
            ||((Word)obj).getHinshi().equals(this.getHinshi()))
      && ( (((Word)obj).getHinshi1() == null && this.getHinshi1() == null)
            ||((Word)obj).getHinshi1().equals(this.getHinshi1()))
      && ( (((Word)obj).getHinshi2() == null && this.getHinshi2() == null)
            ||((Word)obj).getHinshi2().equals(this.getHinshi2()))
```

```
        && ( (((Word)obj).getHinshi3() == null && this.getHinshi3() == null)
               ||((Word)obj).getHinshi3().equals(this.getHinshi3()))
        && ( (((Word)obj).getKatsuyoKata() == null && this.getKatsuyoKata() == null)
               ||((Word)obj).getKatsuyoKata().equals(this.getKatsuyoKata()))
        && ( (((Word)obj).getKatsuyoKei() == null && this.getKatsuyoKei() == null)
               ||((Word)obj).getKatsuyoKei().equals(this.getKatsuyoKei()))
        && ( (((Word)obj).getGenkei() == null && this.getGenkei() == null)
               ||((Word)obj).getGenkei().equals(this.getGenkei()))
        && ( (((Word)obj).getYomi() == null && this.getYomi() == null)
               || ((Word)obj).getYomi().equals(this.getYomi()))
        && ( (((Word)obj).getHatsuon() == null && this.getHatsuon() == null)
               || ((Word)obj).getHatsuon().equals(this.getHatsuon()))
        )
        {
            return true;
        }

        //そのほかは不一致
        return false;
    }
     (略)
}
```

これによって，オブジェクトの内容が一致しているかどうかを検査することができる．

(5) リストの長さを測る

ソート前の状態を確認する．それと併せて，出現した語数を計算しよう．ソート前のリストを表示して，出現語数を測るプログラムリストの例をリスト 6 に示す．

```
//今のリストのエントリをすべて表示する
int sum = 0;
for (int i = 0; i < list.size(); i++)
{
    //System.out.println(list.get(i).getWord().getHyousoukei()
                                                    + ":" + list.get(i).getCount());
    sum += list.get(i).getCount();
}
```

リスト 6　出現語数を計算する

(6) 出現頻度の順にソートする

どの単語が文書の中で最も出現頻度が高いのか，結果を確認しやすいように，出現頻度が高い順にソートする．今回は ArrayList を用いているので，ソートメソッドが用意されている．標準のソートメソッドでは，自然順序で並べ替えがなされるが，今回のオブジェクトでは，その仕組みは適用できないので，オブジェクトの比較ルールを定義する必要がある．

リストのデータをソートするために定義するメソッドの定義をリスト 7 に示す．

```
class WordCompare implements Comparator<WordCount>
{

   @Override
   public int compare(WordCount wc1, WordCount wc2)
   {
      if(wc1.getCount() < wc2.getCount())
         return 1;
      if(wc1.getCount() == wc2.getCount())
         return 0;
      if(wc1.getCount() > wc2.getCount())
         return -1;
      return 0;
   }

}
```

リスト 7　リストのソートのためのメソッドの例

このソートメソッドが定義されたクラスのオブジェクトをソートメソッドに渡すことで，ソートされる．

(7) 単語の重みを与え，出力する

各単語の出現回数を全単語数で割る．与えられた重みの結果をファイルに出力する．

（単語）	（出現回数）	（tf 値）
天然記念物	16	0.071111
県	11	0.048889
両生類	7	0.031111

リスト 8 term frequency のデータを出力する形式の例

ここで，「人間が見てわかる形式」とは，テキストエディタ等で開いて確認できる形式を意味している．リスト 8 のように，1 行が一単語の状況を表していると，人間が見て確認することができる．

「コンピュータで処理可能な形式」とは，このファイルを読み込んで，数値データとしてコンピュータが処理することができる形式を意味している．リスト 8 では，

「単語」¥t（タブ）「数字」¥t（タブ）「数字」¥n

最終的には，ファイルに出力するが，さしあたり，標準出力画面に表示させて確認する．プログラムの例をリスト 9 に示す．

```
//tf の結果をファイルに保存する
try
{
   String outputFilename = "data¥¥001tf.txt";
   FileWriter fw = new FileWriter(outputFilename);
   System.out.println(outputFilename);

   for (int i = 0; i < list.size(); i++)
   {
     //System.out.println(fileal.get(i).word.getHyousoukei() + ":"
                                                          + fileal.get(i).getCount());

      list.get(i).tf = (double) list.get(i).getCount() / (double) sum;
```

```
        fw.write(list.get(i).getWord().getHyousoukei() + "¥t"
                 + list.get(i).getCount() + "¥t"
                + String.format("%.10f", list.get(i).tf + "¥n");
    }
    fw.close();
}
catch (IOException ex)
{
    ex.printStackTrace();
}
```

リスト 9　tf による重みの表示と出力

5.3 100 件のテキストファイルに対して，同様の処理を行う

ここまでの過程で作成したプログラムを変更して, 100 件のテキストファイルを対象に，連続で処理できるようにする.

そのために，TermFrequency の main メソッドで処理している内容を，外部から呼び出すメソッドにする．その際に，出力ファイルと入力ファイルを指定して実行できるようにする．処理で生成するリストはクラスのフィールドにする.

```
public class TermFrequency
{
    ArrayList<TfCount> list = new ArrayList<TfCount>();

    //public static void main(String[] args)
    public void tf(String inputFilename, String outputFilename)
    {
        （ここに main 関数に書いていた処理が来る）

        （ファイル名はメソッドで指定されることにする）
        //String inputFilename = "data¥¥001.txt";
        //String outputFilename = "data¥¥001tf.txt";
    }
}
```

リスト 10　100 件分のファイル名生成

さらに，TermFrequency クラスをインスタンス化して実行できるようにする.

```
package nlp;
public class NaturalLanguageProcessing
{
    static public void main(String args[])
    {
        System.out.println("TF 導出");
        TermFrequency[] tf= new TermFrequency[100];

        for(int i=1; i <= 100;i++)
        {
            tf[i-1] = new TermFrequency();

            String inputFileName = "data¥¥" + String.format("%03d", i) + ".txt";
            String outputFileName = "data¥¥" + String.format("%03d", i) + "tf.txt";

            System.out.println(inputFileName);
            System.out.println(outputFileName);

            tf[i-1].tf(inputFileName,outputFileName);
```

```
        }
    }
}
```

5.4 考察

開発したプログラムが出力する tf による重み付けの結果を評価する．以下に，検討する内容の例を示す.

(1) 正規化によって，文字数の量に依存しない重みになっていることを，自分が注目した単語について確認する.

(2) tf の結果をどのような目的で利用できそうか？

自然言語処理　第 2 回
(Inverse Document Frequency)

1. 概要

単純に文書中に出現する単語の頻度のみによって重み付けを行うと，何度も何度も繰り返し同じ単語が出現する文書の方がより高い重みを持つこととなる．そうすると，要約しているページや簡潔に表現しているページの重みが低くなってしまうという問題が発生する．そこで今回の実験では，文書群として出現頻度が低いが文書の特徴となる，出現頻度が低い単語を重要視する考え方を重み付けに導入する．

2. idf (inverse document frequency)

idf は,「まれに出現する単語は文書群のなかで重要である」という観点に基づく重み付け手法である．たとえば，100 万件の文書の中に「自動車」について書かれているページが 1 万件あった場合と，「スペースシャトル」について書かれているページが 1 千件あった場合では，スペースシャトルという単語は自動車という単語に比べて 10 倍情報を絞り込むことができる．つまり，文書の特定性が高まる．

tf は，1 件の文書中の特徴量であるが，*idf* は，文書群の特徴量であるといえる．

語句 *t* に対する *idf* は式(2)で定義される．

$$idf(t) = \log_{10} \frac{N}{df(t)} + 1 \qquad (2)$$

$df(t)$は全文書中で単語 *t* を含んでいる文書数であり，*N* は対象となる全文書数である．

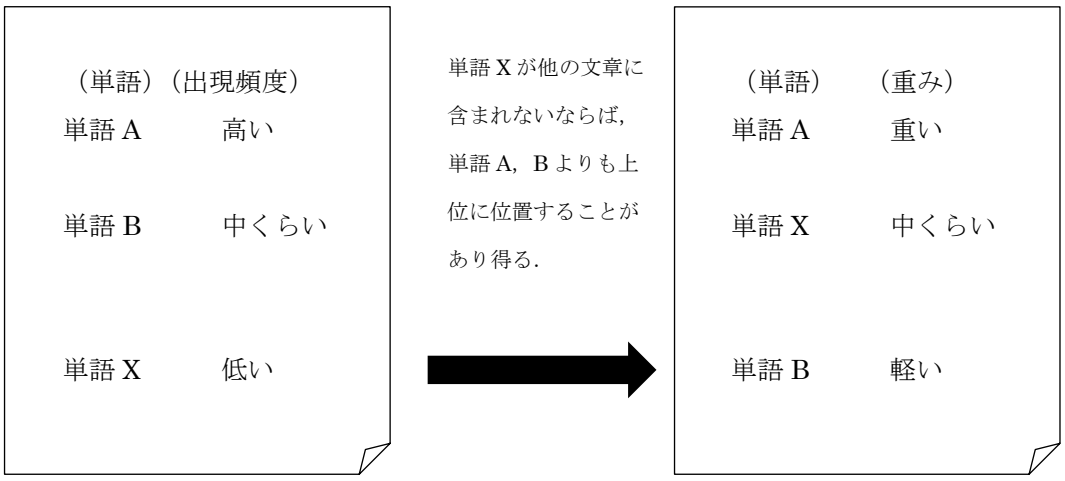


図 45　出現がまれな語句が文書を特長づけるイメージ

たとえば，1000 件の文書に，「自動車」を含んだ文書が 10 件あった場合には，$df(\text{自動車}) = 10$，N=1000 であることから，$idf(t) = 3$である．他方，「トゥクトゥク」

を含んだ文書が 1 件あった場合には，$df(\text{トゥクトゥク}) = 1$，N=1000 であることから $idf(t) = 4$である．

式(2)の定義により，全文書に出現するような単語は重みが 1 となり，1 件の文書にのみ出現する語の重みが最大となる．この重み付けによって，この文書群のなかで使われていることがまれである語句が明らかになる．

3. 実験

3.1 語，df，idf を格納するクラスとデータ型

形態素解析した結果の語（Word クラス）と，文書に出現した回数（DF）をカウントするための Integer クラスのオブジェクト，DF の値から導出する IDF の値を格納する Double クラスのオブジェクトが必要である．さらにこれらを連続して格納するためのデータ型が必要である．これらを定義する．

```
class DfCount extends WordCount
{
   private Double idf;

    （必要とするコンストラクタを追加）
}
```

3.2 df を算出する

算出した tf ファイルを読み込み，文書群に単語が出現する文書件数を算出する．

単語と df 値の対応がわかるデータファイルを出力する．例として，リスト 11 に，document frequency を出力するデータ形式を示す．df の値は，ファイル群から算出するので，100 件のファイル群全体でリスト 11 のような 1 件の重みリストファイルを出力する．

```
（単語）（df）（idf）
県              37   1.431798276
・・・
天然記念物   2   2.69890004
・・・
両生類          1   3.00000000
```

リスト 11　document frequency を出力するデータ形式の例

df ファイルの位置づけの例として，10 件の文書に対して tf を計算した結果のファイルを基に，その中にリストアップされている単語から，df を計算してその結果をファイルに出力するイメージを図 46 に示す．

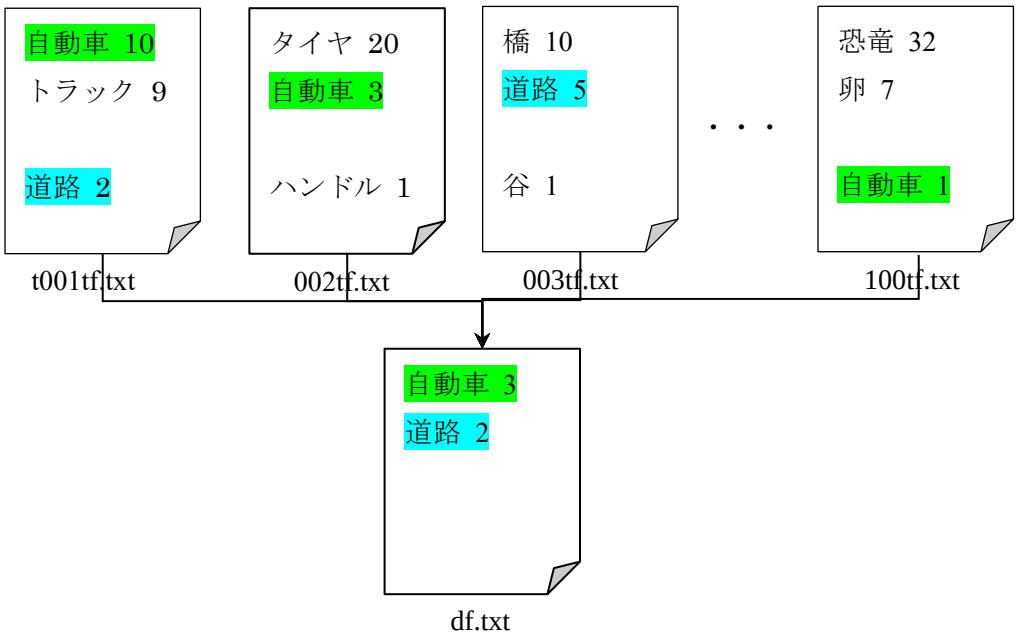


図 46　tf の結果から df を計算する

この時に，df の値を格納するために，データを格納するために適したクラスを設計し，データ型を選んでを使う．

100 件のファイルを開き，それぞれのファイルに記載されている 1 行ごとのデータをリストに取り込んでいく手順を確認する．

(1) DocumentFrequency クラス

DF をカウントして，IDF を格納する処理をするためのクラスを定義する．おおよそ，以下のようになるだろうか．

```
public class DocumentFrequency
{
    //DF をカウントするためのデータ格納領域の定義


    //DF の元になる TF を受ける
    TermFrequency tf[];
    DocumentFrequency(TermFrequency[] tf)
    {
        this.tf = tf;
    }

    public void df(String outputFilename)
    {
         （ここに処理を定義する）
        //TF100 ファイル分について繰り返す

            //TF1 件に含まれる語の分だけ繰り返し

                //DF のリストの中にエントリがあるか調べる

                //リストにエントリが無かったときは，新しい語としてリストに追加する

        //ソートする

        //df,idf の結果をファイルに保存する

    }
}
```

リスト 12　df クラスの例

(2) tf ファイルにリストアップされている単語を検出する

TF にリストアップされている単語は，文書に出現した単語である．DF を計数するデータのリストにリストアップされていたら，その単語のカウントを 1 増加させる．DF のリストにエントリがなかったら追加して，出現回数を 1 としておく．

(3) 100 個のファイルに対して，繰り返す

100 個の tf のリストデータを対象に，文書に出現する回数をカウントする．100 個のデータ群を対象にしているため，最大値は 100 になる．最小値は 1 になる．

(4) df 値でソートする

df の順番でソートする．

3.3 idf を計算して，出力する

idf の定義式に従って，idf を計算する．単語と idf 値の対応がわかるデータファイルを出力する．リスト 13 では，df に続けて idf の値を表示するような出力ファイルフォーマットとしたときの例である．

```
（単語）（df）（idf）
県              37      1.431798276
・・・
天然記念物      2       2.698970
・・・
両生類          1       3.000000
```

リスト 13　df と idf の値を出力するデータ形式の例

```
package nlp;
public class NaturalLanguageProcessing
{
   static public void main(String args[])
   {
      System.out.println("TF 導出");
      （略）

      System.out.println("DF 導出");
      DocumentFrequency df = new DocumentFrequency(tf);
      String outputFilename = "data¥¥df.txt";
      df.df(outputFilename);
   }
}
```

3.4 idf の意味するところを考える

df および，idf はどのような重みを表しているであろうか？idf の結果を用いると，どのようなことがわかるだろうか？

自然言語処理　第 3 回
(tf-idf)

1. 概要

文書に含まれる頻度の高さから特徴を表す重みづけとして *tf* を与え，文書群で出現する頻度が低い，稀である特徴を表す値として *idf* を算出した．それぞれ，文書と文書群から導出した値であるので，互いの値を組み合わせて，文書群に含まれる文書の特徴を表す重みづけとして，*tf-idf* を与える．

2. tf-idf(term frequency - inverse document frequency)

idf 単独では，文書群での特徴量を表す重みになり，このままでは，文書の特徴量となるような，文書の重み付けに生かすことはできない．そこで，文書群での特徴量を，各文書での特徴量にするために，*tf* で算出した重み付けに *idf* の結果を反映させて，*tf-idf* による重み付けを算出する．

これによって，文書中の出現回数としては頻度が小さかった語句でも，他の文書に含まれていなかった場合には，この文書を特徴づける語句となる可能性がある．したがって，出現頻度が低い語句であっても，この文書を特徴づける重みとして，有効に作用することが予想される．

tf と *idf* を掛け合わせた *tf-idf* は式(3)であらわされる．

$$w_{tf-idf}{}_t^d = \frac{tf(t,d)}{\sum_{s \in d} tf(s,d)} \times \left(\log_{10} \frac{N}{df(t)} + 1\right) \tag{3}$$

一般に，*tf* による重み付けによって文書の網羅性を評価し，*idf* による重み付けによって文書の特定性を評価する．この二側面からの重み付けを行うことにより，統計的に文書の順位付けが可能となる．

テキストファイルから，tf，df，idf，tf-idf までの一連で出力するファイルの位置づけを図 47 に示す．

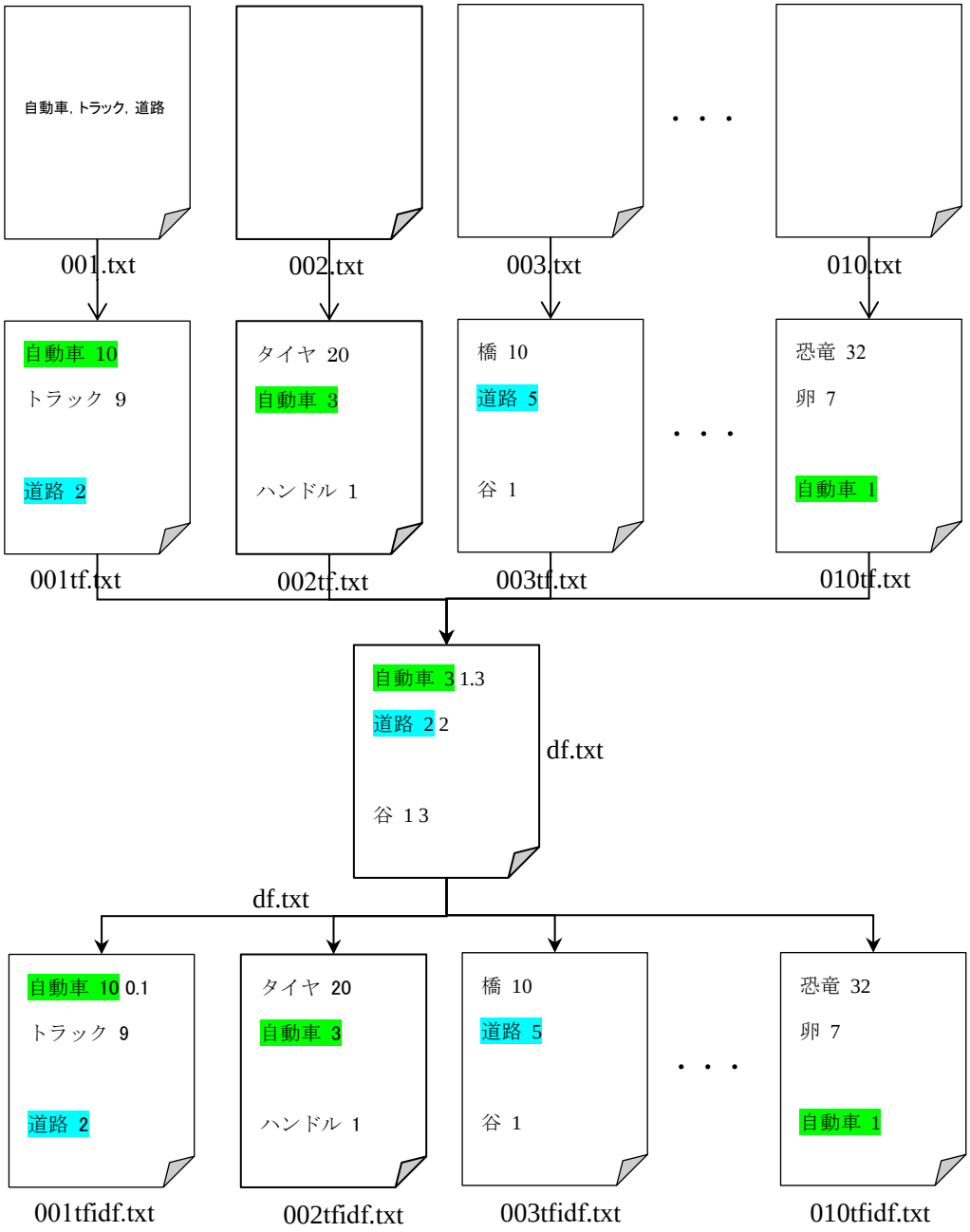


図 47　tf，df，idf，tf-idf，それぞれの位置づけ

3. 実験

3.1 Tf-idf のデータを格納するクラスとデータ型

語と，tf 値，tf-idf 値を持つためのクラスを用意する．このクラスを格納するデータ構造に格納していく．ここでは，TfIdfObj クラスとする．

3.2 *tf-idf* を計算する

各文書の *tf* の値に *idf* を乗じて，*tf-idf* による重み付けを与える．単語と *tf-idf* 値の

対応がわかるデータファイルを出力する．リスト 14 では，*tf* に続けて，*tf*×*idf* を計算した値を表示するような出力ファイルフォーマットとしたときの例である．このファイルは，100 件の文書それぞれに対して出力される．

（単語）	（出現回数）（tf）	（tf-idf）
天然記念物	16	0.071111	0.191926
両生類	7	0.031111	0.093333
県	11	0.048889	0.063606

リスト 14　tf に加えて，tf-idf を出力するデータ形式の例

リスト 15　tf-idf 用リストを構成する構造体の例

(1) TfIdf **クラスを用意する**

tf-idf を導出する，tf のオブジェクトと，df のオブジェクトを受け取って，TF-IDF の一連のデータを格納するデータ型を持ち，TF-IDF を計算してソートして結果を出力するメソッドを持つ．

(2) Tf **を求める**

Tf リストの各語について，idf の値を検索し，word，tf，tf×idf をオブジェクトにまとめて，格納する．

(3) tf-idf **値でソートする**

tf-idf の重みを含むリストにおいて，tf-idf の値が大きい順番でソートする．ソートするための Comparator 関数を定義して，ソートする．

(4) tf-idf **値をファイルに出力する**

ソートした結果をファイルに出力する．

(5) 100 **件のファイルについて繰り返す**

3.3 *tf-idf* によって重み付けした結果を考察する

tf，idf，tf-idf で重みづけの関係や変化を観察する．

例えば，想定として，ある単語をキーワードとして文書群から適した文書を検索することを考える．そのとき，どのデータを用いて，どのような基準で適した文書を選ぶだろうか？重みづけを用いて判断するときに，tf で判断したときと，tf-idf で判断したときでは，違いは生じるだろうか？違いが生じる場合があれば，それを具体的に示してみよう．また，その違いが生じた原因は何だろうか？

tf のみで重み付けしたときの検索結果，*idf* のみで重み付けしたときの検索結果，*tf* と *idf* の積で重み付けしたときの検索結果，それぞれの違い，それぞれがどのような結果を意味しているのか，実行結果より考察する．

自然言語処理　第 4 回
（処理結果の改善）

1. 概要

ここまで進めてきた処理は，文書全体を mecab にて形態素解析し，その結果として出力された分かち書きした単語をカウントしたものである．その結果として，文書の特徴を表す量が重みとしてあらわされたが，その結果は，十分満足のいくものであるだろうか？

そこで，結果を分析し，その分析結果に問題点や改善の余地があれば，それを解決する方法を考え，プログラムに反映させる．

2. 改善例

mecab が出力する情報を利用することで，改善に導くことができるかもしれない．mecab が出力する情報を活用した改善のヒントを以下に示す．

以下のすべてを行う必要はないが，結果を改善するための手段として考えてほしい．

2.1 文節ごとの形態素解析結果を利用する

形態素解析した結果の出力を格納したオブジェクトには，形態素解析が文節ごとに処理した結果が含まれる．この結果を利用することで，解析結果の必要な部分へのアクセスが容易に利用できる．

2.2 素性情報を活用する

形態素は，分解した形態素の種類を示しているので，「名詞の場合のみ単語として抽出する」時には，有益な情報である．標準では，リスト 16 で示す種類ごとに定義された数字が定義されている．形態素 ID は

C:¥Program Files(x86)¥MeCab¥dic¥ipadic¥pos-id.def にて定義されている．

助詞が抽出された場合や，名詞に含まれる記号や数値は分析に役にたつだろうか？

その他,間投,*,* 0	助詞,並立助詞,*,* 23	名詞,固有名詞,地域,国 47
フィラー,*,*,* 1	助詞,連体化,*,* 24	名詞,数,*,* 48
感動詞,*,*,* 2	助動詞,*,*,* 25	名詞,接続詞的,*,* 49
記号,アルファベット,*,* 3	接続詞,*,*,* 26	名詞,接尾,サ変接続,* 50
記号,一般,*,* 4	接頭詞,形容詞接続,*,* 27	名詞,接尾,一般,* 51
記号,括弧開,*,* 5	接頭詞,数接続,*,* 28	名詞,接尾,形容動詞語幹,* 52
記号,括弧閉,*,* 6	接頭詞,動詞接続,*,* 29	名詞,接尾,助数詞,* 53
記号,句点,*,* 7	接頭詞,名詞接続,*,* 30	名詞,接尾,助動詞語幹,* 54
記号,空白,*,* 8	動詞,自立,*,* 31	名詞,接尾,人名,* 55
記号,読点,*,* 9	動詞,接尾,*,* 32	名詞,接尾,地域,* 56
形容詞,自立,*,* 10	動詞,非自立,*,* 33	名詞,接尾,特殊,* 57
形容詞,接尾,*,* 11	副詞,一般,*,* 34	名詞,接尾,副詞可能,* 58
形容詞,非自立,*,* 12	副詞,助詞類接続,*,* 35	名詞,代名詞,一般,* 59
助詞,格助詞,一般,* 13	名詞,サ変接続,*,* 36	名詞,代名詞,縮約,* 60
助詞,格助詞,引用,* 14	名詞,ナイ形容詞語幹,*,* 37	名詞,動詞非自立的,*,* 61
助詞,格助詞,連語,* 15	名詞,一般,*,* 38	名詞,特殊,助動詞語幹,* 62
助詞,係助詞,*,* 16	名詞,引用文字列,*,* 39	名詞,非自立,一般,* 63
助詞,終助詞,*,* 17	名詞,形容動詞語幹,*,* 40	名詞,非自立,形容動詞語幹,* 64
助詞,接続助詞,*,* 18	名詞,固有名詞,一般,* 41	名詞,非自立,助動詞語幹,* 65
助詞,特殊,*,* 19	名詞,固有名詞,人名,一般 42	名詞,非自立,副詞可能,* 66
助詞,副詞化,*,* 20	名詞,固有名詞,人名,姓 43	名詞,副詞可能,*,* 67
助詞,副助詞,*,* 21	名詞,固有名詞,人名,名 44	連体詞,*,*,* 68
助詞,副助詞／並立助詞／終助詞,*,* 22	名詞,固有名詞,組織,* 45	
	名詞,固有名詞,地域,一般 46	

リスト 16　形態素の種類を表す ID

2.3 品詞の種類に注目する

これまでの説明では，主に名詞を代表として扱ってきたが，名詞以外の品詞では，どのような特徴が現れているだろうか．品詞の種類を分けて計数した結果では，どのような特徴が現れるだろうか？

3. 課題

3.1 重みづけの結果を評価する

tf，df，tf-idf を比較検討して，それぞれの特徴をまとめる．また，それぞれは，どのように活用できそうであるかを考えてみる．

3.2 重みづけの結果の問題点を挙げる

ここまでの処理では，形態素解析した単語をすべて数え上げきたが，その結果は，活用するための結果として好ましいデータであるだろうか？もし，好ましくない部分があるとすれば，どのような点が適切でないだろうか？

3.3 問題点を解決する方法を考える.

重みづけした結果から明らかになった問題点について，解決可能な方法を検討し，改善する．

改善した結果と，改善前の結果と比較してどのように変化したか，確かめる．

参考事項

2020.4.1 **更新**