|  |
| --- |
| 博士課程教育リーディングプログラム  実世界データ循環学リーダー人材育成プログラム |
| データツールファースト |
| Mahout入門 |

|  |
| --- |
| 名古屋大学  2014/03/03 |

* Linuxは、Linux Torvaldsの米国およびその他の国における登録商標あるいは商標です。
* Javaは、Oracle Corporationおよびその子会社・関連会社の米国およびその他の国における登録商標です。
* その他の、記載されている会社名、製品名は、各社の登録商標あるいは商標です。

目次

[1. はじめに 1](#_Toc381573455)

[1.1 背景 1](#_Toc381573456)

[1.2 本書について 1](#_Toc381573457)

[2. 概要 2](#_Toc381573458)

[2.1 機械学習とは 2](#_Toc381573459)

[2.2 Mahoutとは 3](#_Toc381573460)

[2.3 Hadoopによる分散処理 3](#_Toc381573461)

[2.4 mahoutコマンド 4](#_Toc381573462)

[3. レコメンデーション 5](#_Toc381573463)

[3.1 概要 5](#_Toc381573464)

[3.2 ユーザベースのレコメンデーション 6](#_Toc381573465)

[3.2.1 概要 6](#_Toc381573466)

[3.2.2 GenericUserBasedRecommenderを利用した実装例 6](#_Toc381573467)

[3.3 アイテムベースのレコメンデーション 11](#_Toc381573468)

[3.3.1 概要 11](#_Toc381573469)

[3.3.2 GenericItemBasedRecommenderを利用した実装例 11](#_Toc381573470)

[3.3.3 mahoutコマンドによる実行例 13](#_Toc381573471)

[3.3.4 Hadoopを利用した実行例 13](#_Toc381573472)

[3.4 類似性指標 14](#_Toc381573473)

[3.4.1 ユークリッド距離 14](#_Toc381573474)

[3.4.2 ピアソン相関係数 14](#_Toc381573475)

[3.4.3 スピアマン相関係数 15](#_Toc381573476)

[3.4.4 Tanimoto係数 15](#_Toc381573477)

[4. クラスタリング 16](#_Toc381573478)

[4.1 概要 16](#_Toc381573479)

[4.2 階層的クラスタリング 16](#_Toc381573480)

[4.3 k平均法クラスタリング 17](#_Toc381573481)

[4.3.1 概要 17](#_Toc381573482)

[4.3.2 KMeansDriverを利用した実装例 18](#_Toc381573483)

[4.3.3 mahoutコマンドによる実行例 22](#_Toc381573484)

[4.4 キャノピークラスタリング 24](#_Toc381573485)

[4.4.1 概要 24](#_Toc381573486)

[4.4.2 CanopyClustererを利用した実装例 25](#_Toc381573487)

[4.4.3 CanopyDriverを利用した実装例 26](#_Toc381573488)

[4.4.4 mahoutコマンドによる実行例 30](#_Toc381573489)

[5. 分類 37](#_Toc381573490)

[5.1 概要 37](#_Toc381573491)

[5.2 単純ベイズ法 38](#_Toc381573492)

[5.2.1 概要 38](#_Toc381573493)

[5.2.2 Mahoutによる単純ベイズ法の実行例 39](#_Toc381573494)

[5.3 決定木 46](#_Toc381573495)

[5.3.1 概要 46](#_Toc381573496)

[5.3.2 GINI係数 47](#_Toc381573497)

[5.3.3 エントロピー 47](#_Toc381573498)

[5.4 Random Forest 48](#_Toc381573499)

[5.4.1 概要 48](#_Toc381573500)

[5.4.2 mahoutコマンドによる実行例 49](#_Toc381573501)

[5.5 評価 51](#_Toc381573502)

[5.5.1 正解率 51](#_Toc381573503)

[5.5.2 混同行列 51](#_Toc381573504)

[5.5.3 エントロピー行列 51](#_Toc381573505)

[5.5.4 AUC 51](#_Toc381573506)

[6. 参考文献 53](#_Toc381573507)

[7. 変更履歴 54](#_Toc381573508)

# はじめに

## 背景

「ビッグデータ」と呼ばれる、従来のデータベースやアプリケーションでは処理しきれない巨大なデータが、近年脚光を浴びています。現代社会では、センサーが常時計測するデータ、インターネット上を飛び交う情報、映像や画像など、様々なデータが日々大量に発生し、ビッグデータとして蓄積されています。また、行政機関や自治体などから、「オープンデータ」という形で、無償で入手できるものもあります。これらを、人間の手ではなく、コンピュータを使用して分析し、従来では気づかなかった相関を見出したり、推測や検出を自動的に行うことで、研究やビジネスに役立てることができます。

「機械学習」は、そのための技法の一つです。機械学習により、相関分析や分類などを自動で行うことができるため、データ分析の手段として注目されています。たとえば、「Mahout」や「Jubatus」など、機械学習を行うためのライブラリやフレームワークが、オープンソースで公開されています。いずれも複数のコンピュータによる分散処理に対応しており、機械学習を高速に行うことができます。

## 本書について

本書は、Apache Mahoutを使用して機械学習を行うための入門書です。Mahoutに付属するコマンドで機械学習が行えるようになること、MahoutをJavaのライブラリとして利用できるようになることを目的としています。

まず2章では、機械学習およびMahoutの概要について解説します。そして、Mahoutが対象とする機械学習のうち、レコメンデーション、クラスタリングおよび分類に着目し、3章、4章および5章で、それぞれの概要と使用方法を解説します。また、「Hadoop」

による分散処理についても、実際の使用例をいくつか説明します。

本書で対象とするMahoutのバージョンは、2013年7月に公開されたバージョン0.8です。また、Hadoopは、ディストリビューション「CDH(Claudera’s Distribution including Apache Hadoop)」のバージョン 3u6 を使用しています。

# 概要

## 機械学習とは

機械学習は、人工知能の一分野であり、人間が行っている学習能力と同等の機能の一部をコンピュータで実現しようとする技術・手法です。大量のデータをもとに、アルゴリズムに従ってデータを分析し、データの特質を推測するための情報を抽出します。そして、その結果から、新たなデータに対して分類や推測などを行います。

データから何らかの傾向を見出す能力は、機械学習のアルゴリズム次第です。人間は、膨大な知識や経験をもとに、柔軟な判断を行うことができます。ですが、人間が自らデータ分析し、判断を行うためのルールを記述するという手法を、膨大なデータに対して行うことは、現実的ではありません。機械学習では、使用するアルゴリズムが想定しないデータに遭遇すると、誤った分析を行う可能性があるため、対象のデータの特徴や目的を理解した上で、機械学習を使うかどうか判断する必要があります。

機械学習は、様々なところで利用されています。以下にその例を示します。

1. 文字認識、パターン認識

文字やパターンの識別に、ニューラルネットワークなどの機械学習が用いられます。

1. 車の自動運転

道路の状態、方向や速度、位置などから状況を判断し、自動運転を行う際に、機械学習が用いられます。

1. 遺伝子と病気の相関分析

遺伝子の発現量と病気との相関を分析するには、膨大なデータを対象とする必要があり、機械学習が積極的に用いられています。

1. 株価予測

過去の株価データなどから株価を予測するために、機械学習が用いられています。

1. スパムメールの検出

スパムメールの検出に、ベイジアンフィルタなどの機械学習が用いられています。

1. サーバへの不正アクセス検出

通常とは異なるパケットなどの検出に、機械学習が用いられます。

## Mahoutとは

Mahoutは、「Apache Software Foundation」が開発および公開している、オープンソースの機械学習ライブラリです。機械学習の様々なアルゴリズムを、安定かつ高速に動作させることを目的としています。主に、「レコメンデーション(協調フィルタリング)」「クラスタリング」「分類」が実装されています。

Mahoutは「Java」で記述されているため、Javaのライブラリとして提供されます。それ以外に、「mahout」というコマンドも提供されています。mahoutコマンドを用いれば、プログラミングを行うことなく機械学習を行うことも可能です。

ちなみに、Mahoutは「マハウト」と読みます。ヒンズー語で「象使い」という意味です(図 1)。Mahoutが、オープンソースの分散処理環境Hadoopを使ってスケーラブルに動作するよう実装されていることから命名されています。(Hadoopのマスコットは「象」です(図 2))。



図 1Mahoutのロゴ



図 2Hadoopのロゴ

## Hadoopによる分散処理

機械学習では、基本的には人間では処理しきれない大量のデータを対象とします。単一のコンピュータでは処理しきれない場合でも、複数のコンピュータを使用して分散処理を行うことで、現実的な時間で有用な情報を引き出せる可能性が高まります。

Mahoutは、すべてではありませんが、Hadoopを利用して分散処理を行えるよう実装されています。

## mahoutコマンド

Mahoutには、機械学習のアルゴリズムやデータ処理などのライブラリだけでなく、実際に機械学習を行うクラスも含まれます。mahoutコマンドからそれらを使用することで、プログラミングを行うことなく機械学習を行うことができます。mahoutコマンドの基本的な書式は、以下の通りです。

|  |
| --- |
| $ mahout コマンド --input 入力ファイル --output 出力ファイル コマンド固有のオプション… |

mahoutコマンドを実行すると、org.apache.mahout.driver.MahoutDriverクラスのmainメソッドを呼び出します。主なコマンドは、入力ファイルと出力ファイルを必要とします。それぞれ「--input」および「--output」オプションで指定します。

コマンド名を知るには、mahoutコマンドを引数なしで実行します。するとコマンドの一覧が出力されます。コマンドの使用方法を知るには、コマンド名の後に「--help」オプションを指定して実行します。するとコマンドの一覧が出力されます。

mahoutコマンドを実行すると、$MAHOUT\_HOME/conf/driver.classes.default.propsの情報をもとに、登録されているクラス(のmainメソッド)を呼び出します。

(「$MAHOUT\_HOME」は、Mahoutのインストールディレクトリを表しています。)

|  |
| --- |
| #Utils  org.apache.mahout.utils.vectors.VectorDumper = vectordump : Dump vectors from a sequence file to text  org.apache.mahout.utils.clustering.ClusterDumper = clusterdump : Dump cluster output to text  org.apache.mahout.utils.SequenceFileDumper = seqdumper : Generic Sequence File dumper  …  #Clustering  org.apache.mahout.clustering.kmeans.KMeansDriver = kmeans : K-means clustering  org.apache.mahout.clustering.fuzzykmeans.FuzzyKMeansDriver = fkmeans : Fuzzy K-means clustering  org.apache.mahout.clustering.minhash.MinHashDriver = minhash : Run Minhash clustering  org.apache.mahout.clustering.lda.cvb.CVB0Driver = cvb : LDA via Collapsed Variation Bayes (0th deriv. approx)  … |

また、上記に登録されていない場合でも、クラス名を直接指定することで実行が可能です。

|  |
| --- |
| $ mahout クラス名 引数… |

# レコメンデーション

## 概要

レコメンデーションとは、ユーザの嗜好(プリファレンス)を推測して、そのユーザがまだ購入していない商品などのアイテムを推薦(レコメンド)することです。たとえば、Amazonや楽天などのECサイトでは、ある商品のページを閲覧したり、商品を購入する(しようとする)と、その商品に関連する「おすすめの」商品が表示されます。ECサイトには、どのユーザが何を閲覧して購入したかなどのデータが記録されています。それらのデータから機械学習を行って、おすすめの商品を推測し、列挙しています。

レコメンデーションで用いられるアルゴリズムには、「協調フィルタリング」と「コンテンツベースのフィルタリング」があります。協調フィルタリングでは、購入履歴など嗜好を表すデータをもとに推測します。データから嗜好を推測するため、アイテムそのものの知識は必要ありません。アイテムが本でも飛行機でも人でも、同じアルゴリズムで学習・推測を行います。

コンテンツベースのフィルタリングでは、アイテムの属性を解析して推測を行います。たとえば、オライリー社の書籍を購入した人は、オライリー社の他の書籍を購入する可能性があると推測するとき、出版社という属性に基づいて推薦を行っていることになります。コンテンツベースのフィルタリングでは、購入履歴などのデータは必要ありませんが、アイテムの属性によって推薦の判断が異なるため、アイテムの知識が必要となります。たとえば、先ほどの出版社による推薦が、書籍以外にも適用できるとは限りません。たとえば生活雑貨など、個人の嗜好がさほど強くないものに関しては、この推薦が当てはまらないことが多いと考えられます。

Mahoutでは、協調フィルタリングを対象としています。協調フィルタリングには、ユーザベースとアイテムベースのレコメンデーションがあります。それぞれの概要と使用方法を、以降で説明します。

## ユーザベースのレコメンデーション

### 概要

ユーザベースのレコメンデーションでは、ユーザ同士の類似性から推薦を行います。対象のユーザと嗜好が類似するユーザの購入履歴から、まだ購入していないアイテムのプリファレンス(嗜好)を推測し、上位のアイテムを推薦します。

ユーザuが購入したアイテムiのプリファレンスを、ユーザv との類似度をとすると、ユーザベースのレコメンデーションでのプリファレンスの推測値は、以下の式で表すことができます。

アイテムのプリファレンスとそのユーザとの類似度をかけたものの総和です。類似度の高いユーザのプリファレンスが重視されます。類似度の詳細は、3.4節で解説します。

プリファレンスを持たないすべてのアイテムに対して上記の値を求め、上位のアイテムを推薦します。ただ、すべてのアイテムに対して計算を行うと時間がかかるため、実際には、類似度の高いユーザだけで計算を行います。

### GenericUserBasedRecommenderを利用した実装例

Mahoutによるユーザベースのレコメンデーションの実装例を示します。

ユーザベースのレコメンデーションを行うには、GenericUserBasedRecommenderクラスを使用します。Mahoutでは、下記クラスのインスタンスを用意するだけで、ユーザベースのレコメンデーションが可能です。

1. Recommender

レコメンデーションを行うためのインタフェースです。

前述の通り、以降の例ではGenericUserBasedRecommenderを使用します。

1. DataModel

ユーザ、アイテムおよびプリファレンスを保持・参照するためのインタフェースです。

以降の例では、CSV形式のファイルを扱うFileDataModelクラスを使用しています。

1. UserSimilarity

ユーザの類似度を提供するためのインタフェースです。

以降の例では、ピアソン相関係数に基づいて類似度を求めるPearsonCorrelationSimilarityクラスを使用しています。

1. UserNeighborhood

似ているユーザを求めるためのインタフェースです。

以降の例では、最も似ているユーザ数から判断するNearestNUserNeighborhoodクラスを使用しています。

まず、ユーザID、アイテムIDおよびプリファレンス値が格納されたCSV形式の入力データを用意します。

入力データ(ユーザID,アイテムID,プリファレンス値)

|  |
| --- |
| 1,101,5.0  1,102,3.0  1,103,2.5  2,101,2.0  2,102,2.5  2,103,5.0  2,104,2.0  3,101,2.5  3,104,4.0  3,105,4.5  3,107,5.0  4,101,5.0  4,103,3.0  4,104,4.5  4,106,4.0  5,101,4.0  5,102,3.0  5,103,2.0  5,104,4.0  5,105,3.5  5,106,4.0 |

このデータを表で表すと、以下のようになります。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **101** | **102** | **103** | **104** | **105** | **106** | **107** |
| **ユーザ1** | 5.0 | 3.0 | 2.5 |  |  |  |  |
| **ユーザ2** | 2.0 | 2.5 | 5.0 | 2.0 |  |  |  |
| **ユーザ3** | 2.5 |  |  | 4.0 | 4.5 |  | 5.0 |
| **ユーザ4** | 5.0 |  | 3.0 | 4.5 |  | 4.0 |  |
| **ユーザ5** | 4.0 | 3.0 | 2.0 | 4.0 | 3.5 | 4.0 |  |

これを、FileDataModelクラスを用いて読み込みます。以下のように、Fileクラスのインスタンスを引数に指定して、DataModelのインスタンスを生成します。

|  |
| --- |
| DataModel model = new FileDataModel(new File(fileName)); |

次に、UserSimilarityとUserNeighborhoodのインスタンスを生成します。

以下では、近傍の2ユーザを使用してレコメンデーションの計算を行います。

|  |
| --- |
| UserSimilarity similarity = new PearsonCorrelationSimilarity(model);  UserNeighborhood neighborhood = new NearestNUserNeighborhood(2, similarity, model); |

そして、GenericUserBasedRecommenderのインスタンスを生成し、レコメンデーションを実行します。以下では、ユーザ1を対象に、推薦するアイテムを最大5つまで求めます。

|  |
| --- |
| Recommender recommender = new GenericUserBasedRecommender(model, neighborhood, similarity);  List<RecommendedItem> recommendations = recommender.recommend(1, 5); |

プログラムの全体を以下に示します。

|  |
| --- |
| import org.apache.mahout.cf.taste.impl.model.file.\*;  import org.apache.mahout.cf.taste.impl.neighborhood.\*;  import org.apache.mahout.cf.taste.impl.recommender.\*;  import org.apache.mahout.cf.taste.impl.similarity.\*;  import org.apache.mahout.cf.taste.model.\*;  import org.apache.mahout.cf.taste.neighborhood.\*;  import org.apache.mahout.cf.taste.recommender.\*;  import org.apache.mahout.cf.taste.similarity.\*;  import java.io.\*;  import java.util.\*;  public class UserRecommenderSample {  public static void main(String arg[]) throws Exception {  if (arg.length != 2) {  System.err.println(  "Usage: java UserRecommenderSample csvfile id");  System.exit(1);  }  File modelFile = new File(arg[0]);  if (! modelFile.exists()) {  System.err.println("open failed : " + modelFile);  System.exit(1);  }  DataModel model = new FileDataModel(modelFile);  long userid = Long.parseLong(arg[1]);  UserSimilarity similarity = new PearsonCorrelationSimilarity(model);  UserNeighborhood neighborhood =  new NearestNUserNeighborhood(  model.getNumUsers()/2, similarity, model);  long[] neighborhoods = neighborhood.getUserNeighborhood(userid);  System.out.print("Neighborhoods: ");  for (long uid : neighborhoods) {  System.out.print(uid + " ");  }  System.out.println();  Recommender recommender = new GenericUserBasedRecommender(  model, neighborhood, similarity);  List<RecommendedItem> recommendations = recommender.recommend(  userid, 5);  System.out.print("Results: ");  for (RecommendedItem item : recommendations) {  System.out.print(item.getItemID() + ":" + item.getValue() + " ");  }  System.out.println();  }  } |

実行例を以下に示します。

ユーザ1に似ているユーザが 4 と 5 で、推薦するアイテムが 104 と 106 という結果が得られています。

|  |
| --- |
| $ mahout UserRecommenderSample data.csv 1  MAHOUT\_LOCAL is set, so we don't add HADOOP\_CONF\_DIR to classpath.  MAHOUT\_LOCAL is set, running locally  2 28, 2014 1:56:21 午前 org.slf4j.impl.JCLLoggerAdapter warn  警告: No UserRecommenderSample.props found on classpath, will use command-line arguments only  2 28, 2014 1:56:21 午前 org.slf4j.impl.JCLLoggerAdapter info  情報: Creating FileDataModel for file data/intro.csv  2 28, 2014 1:56:21 午前 org.slf4j.impl.JCLLoggerAdapter info  情報: Reading file info...  2 28, 2014 1:56:21 午前 org.slf4j.impl.JCLLoggerAdapter info  情報: Read lines: 21  2 28, 2014 1:56:21 午前 org.slf4j.impl.JCLLoggerAdapter info  情報: Reading file info...  2 28, 2014 1:56:21 午前 org.slf4j.impl.JCLLoggerAdapter info  情報: Read lines: 21  2 28, 2014 1:56:21 午前 org.slf4j.impl.JCLLoggerAdapter info  情報: Processed 5 users  **Neighborhoods: 4 5**  **Results: 104:4.257081 106:4.0**  2 28, 2014 1:56:21 午前 org.slf4j.impl.JCLLoggerAdapter info  情報: Program took 137 ms (Minutes: 0.0022833333333333334) |

## アイテムベースのレコメンデーション

### 概要

アイテムベースのレコメンデーションでは、アイテム同士の類似性から推薦を行います。ユーザの嗜好よりもアイテムの嗜好の方が変化しない(ユーザの嗜好は時間とともに変化する可能性がある)ため、データが多いほどアイテム間の類似度は収束するという仮定に基づいています。アイテムベースのレコメンデーションの場合、ユーザuのアイテムiのプリファレンスの推測値は、以下の式で表されます。

ここでは、アイテムiとjの類似度です。

### GenericItemBasedRecommenderを利用した実装例

Mahoutによるアイテムベースのレコメンデーションの実装例を示します。

アイテムベースのレコメンデーションを行うには、GenericItemBasedRecommenderクラスを使用します。3.2.2のGenericUserBasedRecommenderによる実装と、以下の点を除けばほぼ同じ構成で動作します。

1. UserSimilarityではなくItemSimilarityを使用する

ItemSimilarityはアイテムの類似度を提供するためのインタフェースです。

前述のPearsonCorrelationSimilarityはItemSimilarityを実装しています。

1. UserNeighborhoodが不要

プログラムを以下に示します。

3.2.2のプログラムとほぼ同じですが、UserNeighborhoodが不要な分、コードが短くなっています。

|  |
| --- |
| import org.apache.mahout.cf.taste.impl.model.file.\*;  import org.apache.mahout.cf.taste.impl.recommender.\*;  import org.apache.mahout.cf.taste.impl.similarity.\*;  import org.apache.mahout.cf.taste.model.\*;  import org.apache.mahout.cf.taste.recommender.\*;  import org.apache.mahout.cf.taste.similarity.\*;  import java.io.\*;  import java.util.\*;  public class ItemRecommenderSample {  public static void main(String arg[]) throws Exception {  if (arg.length != 2) {  System.err.println(  "Usage: java ItemRecommenderSample csvfile id");  System.exit(1);  }  File modelFile = new File(arg[0]);  if (! modelFile.exists()) {  System.err.println("open failed : " + modelFile);  System.exit(1);  }  DataModel model = new FileDataModel(modelFile);  long userid = Long.parseLong(arg[1]);  ItemSimilarity similarity = new PearsonCorrelationSimilarity(model);  Recommender recommender = new GenericItemBasedRecommender(  model, similarity);  List<RecommendedItem> recommendations = recommender.recommend(  userid, 5);  System.out.print("Results: ");  for (RecommendedItem item : recommendations) {  System.out.print(item.getItemID() + ":" + item.getValue() + " ");  }  System.out.println();  }  } |

実行例を以下に示します。

ユーザ1に推薦するアイテムが 104という結果が得られています。

|  |
| --- |
| $ mahout ItemRecommenderSample data.csv 1  ...  Results: 104:5.0 |

### mahoutコマンドによる実行例

mahoutコマンドで、アイテムベースのレコメンデーションを実行できます。

実行例を以下に示します。

|  |
| --- |
| $ mahout recommenditembased --input data.csv --output data.out \  --similarityClassname SIMILARITY\_PEARSON\_CORRELATION |

アイテムベースのレコメンデーションを行うためのコマンド名は「recommenditembased」です。--inputオプションでCSV形式の入力データファイル、--outputオプションで結果の出力先となるディレクトリ、「--similarityClassname」で後述の類似度を指定します。

(org.apache.mahout.math.hadoop.similarity.cooccurrence.measures.VectorSimilarityMeasuresに、類似度の名前がenum型で登録されています。)

上記を実行すると、出力ディレクトリの下にpart-r-00000というテキストファイルが作成されます。

|  |
| --- |
| $ cat data.out/part-r-00000  1 [104:3.9258494]  3 [102:3.2698717]  4 [102:4.7433763] |

ユーザ1にはアイテム104、ユーザ3および4にはアイテム102を推薦するという結果が得られています。

recommenditembasedを実行すると、temp ディレクトリを作成し、その下にファイルをいくつか作成します。次に実行する際、tempディレクトリにそれらのファイルがあるとエラーになる(例外が発生する)ため、削除してから実行するようにしてください。

### Hadoopを利用した実行例

mahoutコマンドの recommenditembasedは、Hadoopによる分散処理に対応しています。環境変数「MAHOUT\_LOCAL」が設定されていなければ、Hadoopを利用します。基本的な書式は同じですが、ファイルを、Hadoopの分散ファイルシステム「HDFS(Hadoop Distributed File System)」に置く必要があります。

|  |
| --- |
| $ hadoop fs -put data.csv mahout-data  $ mahout recommenditembased --input mahout-data/data.csv --output mahout-data/data.out --similarityClassname SIMILARITY\_PEARSON\_CORRELATION  $ hadoop fs -cat mahout-data/data.out/part-r-00000  1 [104:3.9258494]  3 [102:3.2698717]  4 [102:4.7433763] |

Hadoopによる分散処理を行っても、単一マシンで計算した3.3.3と同じ結果になります。

## 類似性指標

類似度には様々な種類があります。代表的なものを以下に示します。

下記の実装クラスはいずれも、org.apache.mahout.cf.taste.impl.similarityにあります。

### ユークリッド距離

ユークリッド距離は、多次元空間にある2点間の距離です。2点の座標がそれぞれ(,,…)と(,,…)のとき、ユークリッド距離は以下の式で表されます。

=

たとえば、ユーザの類似度を求めるには、ユーザuのアイテムiのプリファレンスを、ユーザvのアイテムiのプリファレンスをとして、上記の式を用います。

0から1の間の値に正規化したい場合は、下記の式を用います。

ユークリッド距離による類似度の実装クラスはEuclideanDistanceSimilarityです。

### ピアソン相関係数

ピアソン相関係数は、2つの変数の相関を計るための指標です。-1から1の間の値を取り、完全に相関する場合は1、逆相関の場合は-1になります。式は以下の通りです。

は、の平均を表します。

ピアソン相関係数による類似度の実装クラスはPearsonCorrelationSimilarityです。

### スピアマン相関係数

スピアマン係数算出するための式は、3.4.2のピアソン相関係数と同じです。ただし、各値を、プリファレンスではなく順位に置き換えます。そのため、式は以下のようになります。

スピアマン相関係数による類似度の実装クラスはSpearmanCorrelationSimilarityです。

### Tanimoto係数

Tanimoto係数は、2つの集合の類似度を計る指標です。AとBの2つの集合があるとき(図 3)、Aのアイテム数を、Bのアイテム数を、AとBの共通集合のアイテム数をとすると、以下の式で表されます。

たとえば、ユーザの類似度を求めるには、プリファレンスを持つアイテムを集合の要素として、上記の式を用います。

Tanimoto係数による類似度の実装クラスはTanimotoCoefficientSimilarityです。

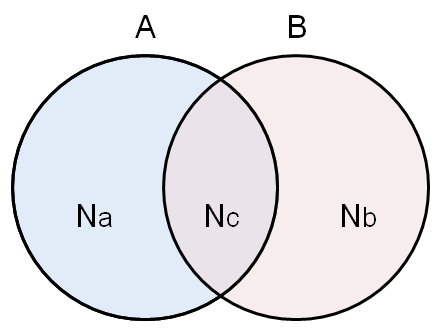


図 3Tanimoto係数を表すベン図

# クラスタリング

## 概要

クラスタリングとは、複数のアイテムを、類似度に従っていくつかのグループに分けることです。分類方法が決まっていないデータに対してクラスタリングを行うことで、何らかのパターンを見出したり、分類のきっかけを得ることができます。また、データを大まかに把握する手助けにもなります。たとえば、購買履歴をもとに、クラスタリングによって顧客を分析し、マーケティングに利用することができます。

クラスタリングにも、様々なアルゴリズムがあります。ここでは、グループにまとめる処理を繰り返してグループの階層を作り上げる「階層的クラスタリング」と、特定のクラスタ数に分ける「非階層型クラスタリング」である「k平均法クラスタリング」と「キャノピークラスタリング」について解説します。

## 階層的クラスタリング

階層的クラスタリングは、もっとも距離の近い2つのクラスタを1つにすることを繰り返すことで、クラスタの階層を作成するクラスタリングの手法のひとつです(図 3)。アルゴリズムを以下に示します。

1. 個々のアイテムをそれぞれクラスタとする。
2. クラスタ間の距離を計算し、もっとも近い2つのクラスタを1つにまとめる。
3. 最終的にクラスタが1つになるまで、2.を繰り返す。

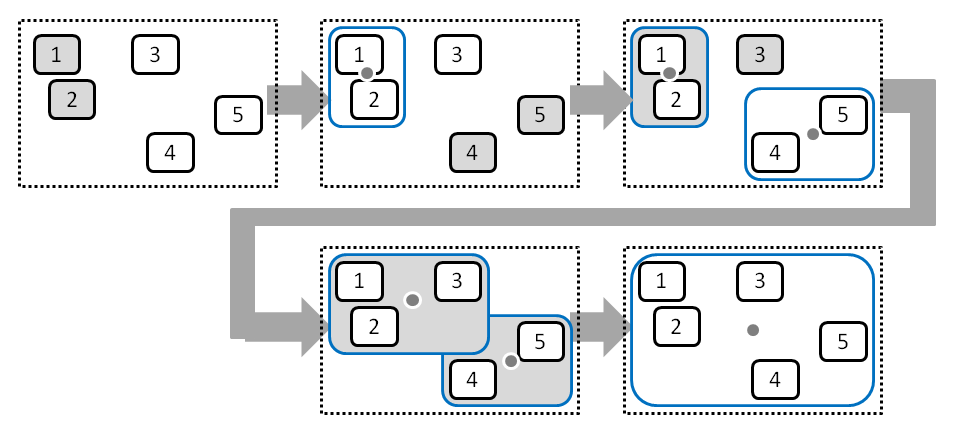


図 4階層的クラスタリングの処理

ここで、クラスタの位置(重心)は、クラスタ内のアイテムの位置の平均です。

また、クラスタ間の距離測度には、3.4節の類似度と同様のものを使用します。Mahoutでは、それぞれの実装クラスがorg.apache.mahout.common.distanceにあります。

階層的クラスタリングで得られた結果を「デンドログラム」で表現すると、より解釈しやすくなります。デンドログラムは、各アイテムがクラスタにまとめられていく過程をツリー状に表したものです(図 4)。クラスタの相対的な距離が可視化されるため、クラスタ間の比較に適しています。また、クラスタとしてまとめられる様子も理解しやすくなります。

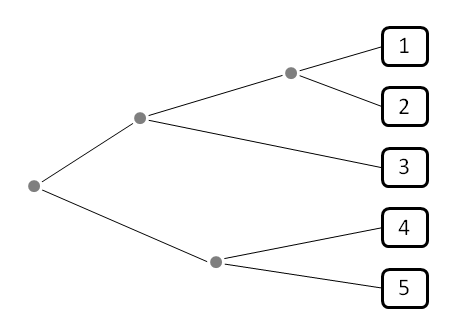


図 5デンドログラムの例

## k平均法クラスタリング

### 概要

階層的クラスタリングの場合、すべてのクラスタ間の距離を計算する必要があります。特に、クラスタをまとめたときには再計算が必要なため、計算量が非常に大きくなります。また、どの階層を使うのか判断が必要になることも、問題点として挙げられます。

k平均法クラスタリングは、非階層的クラスタリングの手法のひとつです。あらかじめ生成するクラスタ数を決めておき、各アイテムをその数のクラスタに割り当てます(図 5)。アルゴリズムは以下の通りです。

1. クラスタの中心点となる重心を、ランダムにk個配置する。
2. 各アイテムを、重心がもっとも近いクラスタに割り当てる。
3. クラスタに属するアイテムの平均を、新たな重心とする。
4. 重心が移動しなくなるまで、2.～3.を繰り返す。

結果が得られるまでの処理回数は、階層的クラスタリングと比較すると非常に小さくなります。また、指定したk個のクラスタへ、明確に分けることができます。

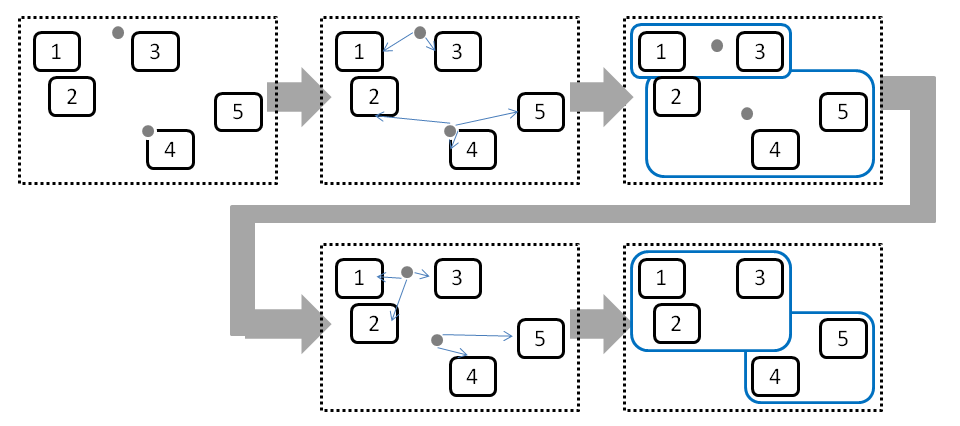


図 6k平均法クラスタリングの処理

### KMeansDriverを利用した実装例

Mahoutによるk平均法クラスタリングの実装例を示します。ここでは、2次元に配置された10個の点をクラスタに分けます。

k平均法クラスタリングを行うには、KMeansDriverクラスを使用します。KMeansDriverでは、HadoopのSequenceFile(シーケンスファイル)形式の入力データと、クラスタの重心の初期値が必要です。入力データがdouble型の2次元配列の場合、以下の手順でシーケンスファイルに出力します。

|  |
| --- |
| Configuration conf = new Configuration();  FileSystem fs = FileSystem.get(conf);  double[][] points = {{0, 1}, …}; // 入力データ  List<Vector> pointsList = createVectors(points); // Vector型のリストに変換  Path path = new Path("kminput" + File.separator + "points" +  File.separator + "file1");  SequenceFile.Writer pwriter = new SequenceFile.Writer( // シーケンスファイルを生成  fs, conf, path, LongWritable.class, VectorWritable.class);  VectorWritable vw = new VectorWritable();  long num = 0;  for (Vector point : pointsList) {  vw.set(point);  pwriter.append(new LongWritable(num++), vw); // 番号とVectorを追加  }  pwriter.close(); |

クラスタの重心の初期値の場合は、KlusterクラスのIDをkey、Klusterクラスのインスタンスをvalueとするシーケンスファイルとして出力します。

|  |
| --- |
| double[][] clusters = {{0, 1}, …}; // クラスタの重心の初期値  List<Vector> clustersList = createVectors(clusters); // Vecotr型のリストに変換  Path path = new Path("kminput" + File.separator + "clusters" +  File.separator + "part-00000");  SequenceFile.Writer writer = new SequenceFile.Writer( // シーケンスファイルを生成  fs, conf, path, Text.class, Kluster.class);  for (int i = 0; i < clustersList.size(); i++) {  Vector v = clustersList.get(i);  Kluster cluster = new Kluster(v, i, new EuclideanDistanceMeasure());  writer.append(new Text(cluster.getIdentifier()), cluster);  }  writer.close(); |

そして、KMeansDriverのrunメソッドを呼び出します。引数に、入力データ、クラスタの重心の初期値、出力ディレクトリをそれぞれPathで指定した後、距離測度にユークリッド距離を使用するEuclideanDistanceMeasureクラスのインスタンスを指定しています。その後は、重心が一定になったと判断するための閾値(0.001)、最大反復回数(10)、クラスタリングの実施(true)などを指定しています。

|  |
| --- |
| KMeansDriver.run(conf, new Path("kminput" + File.separator + "points"),  new Path("kminput" + File.separator + "clusters"), new Path("kmoutput"),  new EuclideanDistanceMeasure(), 0.001, 10, true, 0.0, false); |

プログラムの全体を以下に示します。

|  |
| --- |
| import org.apache.hadoop.conf.\*;  import org.apache.hadoop.fs.\*;  import org.apache.hadoop.io.\*;  import org.apache.mahout.clustering.classify.\*;  import org.apache.mahout.clustering.kmeans.\*;  import org.apache.mahout.common.\*;  import org.apache.mahout.common.distance.\*;  import org.apache.mahout.math.\*;  import java.io.\*;  import java.util.ArrayList;  import java.util.List;  public class KMeansSample {  public static final String OUTPUTDIR = "kmoutput";  public static final String BASEDIR = "kminput";  public static final String POINTSDIR = BASEDIR + File.separator + "points";  public static final String CLUSTERSDIR = BASEDIR + File.separator + "clusters";  public static final double[][] points = {  {0, 1}, {1, 1}, {2, 1}, {1, 9}, {2, 9}, {2, 10},  {9, 5}, {10, 6}, {11, 6}, {10, 7},  };  public static final double[][] clusters = {  {2, 2}, {5, 5}, {8, 8},  };    public static List<Vector> createVectors(double[][] points) {  List<Vector> vList = new ArrayList<Vector>();  for (int i = 0; i < points.length; i++) {  Vector v = new RandomAccessSparseVector(points[i].length);  v.assign(points[i]);  vList.add(v);  }  return vList;  }    public static void main(String args[]) throws Exception {  List<Vector> pointsList = createVectors(points);  List<Vector> clustersList = createVectors(clusters);  File dir = new File(BASEDIR);  if (! dir.exists()) {  dir.mkdir();  }  dir = new File(POINTSDIR);  if (! dir.exists()) {  dir.mkdir();  }  Configuration conf = new Configuration();  FileSystem fs = FileSystem.get(conf);  SequenceFile.Writer pwriter = new SequenceFile.Writer(  fs, conf, new Path(POINTSDIR + File.separator + "file1"),  LongWritable.class, VectorWritable.class);  VectorWritable vw = new VectorWritable();  long num = 0;  for (Vector point : pointsList) {  vw.set(point);  pwriter.append(new LongWritable(num++), vw);  }  pwriter.close();    SequenceFile.Writer cwriter = new SequenceFile.Writer(  fs, conf, new Path(CLUSTERSDIR + File.separator + "part-00000"),  Text.class, Kluster.class);  for (int i = 0; i < clustersList.size(); i++) {  Vector v = clustersList.get(i);  Kluster cluster = new Kluster(v, i, new EuclideanDistanceMeasure());  cwriter.append(new Text(cluster.getIdentifier()), cluster);  }  cwriter.close();  Path output = new Path(OUTPUTDIR);  HadoopUtil.delete(conf, output);  KMeansDriver.run(conf, new Path(POINTSDIR), new Path(CLUSTERSDIR),  output, new EuclideanDistanceMeasure(),  0.001, 10, true, 0.0, false);  SequenceFile.Reader reader = new SequenceFile.Reader(fs,  new Path(OUTPUTDIR + File.separator + Kluster.CLUSTERED\_POINTS\_DIR +  File.separator + "part-m-00000"), conf);  IntWritable key = new IntWritable();  WeightedVectorWritable value = new WeightedVectorWritable();  while (reader.next(key, value)) {  System.out.println(key + " : " + value);  }  reader.close();  }  } |

実行例を以下に示します。

10個の点が3つのクラスタに分けられていることがわかります。

|  |
| --- |
| $ mahout KMeansSample  ...  0 : 1.0: [1:1.000]  0 : 1.0: [1.000, 1.000]  0 : 1.0: [2.000, 1.000]  1 : 1.0: [1.000, 9.000]  1 : 1.0: [2.000, 9.000]  1 : 1.0: [2.000, 10.000]  2 : 1.0: [9.000, 5.000]  2 : 1.0: [10.000, 6.000]  2 : 1.0: [11.000, 6.000]  2 : 1.0: [10.000, 7.000] |

### mahoutコマンドによる実行例

mahoutコマンドでも、k平均法クラスタリングを実行できます。

実行例を以下に示します。

|  |
| --- |
| $ mahout kmeans --input kminput/points --output kmoutput2 \  --clusters kminput/clusters \  --distanceMeasure \  org.apache.mahout.common.distance.EuclideanDistanceMeasure \  --convergenceDelta 0.001 --numClusters 3 --maxIter 10 --clustering |

k平均法クラスタリングを行うためのコマンド名は「kmeans」です。--inputオプションでは4.3.2で作成した入力データ、「--clusters」オプションでは同様に4.3.2で作成したクラスタの重心の初期値を指定します。また、「--distanceMeasure」オプションで距離測度、「--convergenceDelta」オプションで重心が一定と判断するための閾値、「--numClusters」オプションでクラスタ数、「--maxIter」オプションで最大反復回数、「--clustering」でクラスタリングの実施、を指定しています。

上記を実行すると、4.3.2と同様、シーケンスファイルが出力されます。シーケンスファイルをテキストに変換するには、mahoutの「seqdumper」コマンドを用います。

|  |
| --- |
| $ mahout seqdumper --input kmoutput2/clusteredPoints  ...  Key: 2: Value: 1.0: [1:1.000]  Key: 2: Value: 1.0: [1.000, 1.000]  Key: 2: Value: 1.0: [2.000, 1.000]  Key: 4: Value: 1.0: [1.000, 9.000]  Key: 4: Value: 1.0: [2.000, 9.000]  Key: 4: Value: 1.0: [2.000, 10.000]  Key: 9: Value: 1.0: [9.000, 5.000]  Key: 9: Value: 1.0: [10.000, 6.000]  Key: 9: Value: 1.0: [11.000, 6.000]  Key: 9: Value: 1.0: [10.000, 7.000]  Count: 10 |

4.3.2と同じ結果が得られています。

## キャノピークラスタリング

### 概要

k平均法によるクラスタリングを用いると、階層的クラスタリングよりも高速に処理できます。ですが、実世界のデータを対象とする場合、事前にクラスタ数を決めることは、非常に困難です。また、最初に置いた重心の位置によって結果が左右されるという問題もあります。

キャノピークラスタリング(もしくは「キャノピー生成」)は、2つの距離を閾値として用いることで、高速にクラスタリングを行う手法です(図 6)。閾値および(ただし)が与えられたとき、以下のアルゴリズムにより、クラスタの重心を推定します。

1. すべてのアイテムが登録されたリストを用意する。
2. リストの中からアイテムを1つ選ぶ。選んだアイテムから距離以内のアイテムをリストから削除し、クラスタを生成する。
3. 距離以内のアイテムの平均を、生成したクラスタの重心とする。
4. リストが空になるまで、2.～3.を繰り返す。

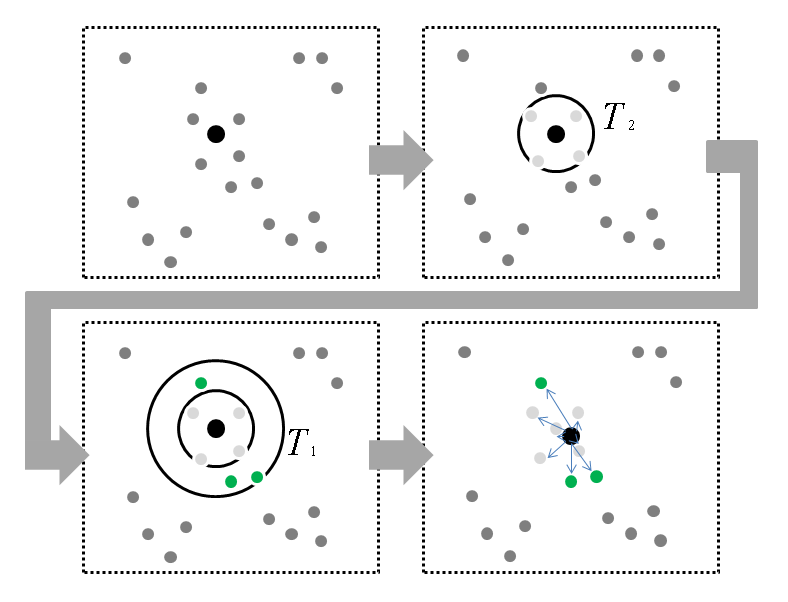


図 7キャノピークラスタリングの処理

キャノピークラスタリングで得られる結果は、クラスタの重心です。この結果を、k平均法クラスタリングの重心の初期位置として使用すると、重心をランダムに置くよりも精度の高い結果が得られます。

ちなみに、キャノピー(canopy)とは、ベッドなどの装飾用の天蓋・覆いのことです。

### CanopyClustererを利用した実装例

Mahoutによるキャノピークラスタリングの実装例を示します。4.3.2と同様、2次元に配置された10個の点をクラスタに分けます。

ここでは、CanopyClustererクラスを使用します。CanopyClustererでは、シーケンスファイルではなく、Vectorクラスのリストを扱います。クラスタリングの結果は、Canopyクラスのリストになります。

下記のように、createCanopiesメソッドで実行します。引数に、入力データ(Vectorクラスのリスト)、距離測度、およびです。

|  |
| --- |
| List<Vector> pointsList = createVectors(points);  List<Canopy> canopiesList = CanopyClusterer.createCanopies(  pointsList, new EuclideanDistanceMeasure(), 8.0, 4.0); |

プログラムの全体を以下に示します。

|  |
| --- |
| import org.apache.mahout.clustering.canopy.\*;  import org.apache.mahout.common.distance.\*;  import org.apache.mahout.math.\*;  import java.util.ArrayList;  import java.util.List;  public class CanopyClustererSample {  public static final double[][] points = {  {0, 1}, {1, 1}, {2, 1}, {1, 9}, {2, 9}, {2, 10},  {9, 5}, {10, 6}, {11, 6}, {10, 7},  };    public static List<Vector> createVectors(double[][] points) {  List<Vector> vList = new ArrayList<Vector>();  for (int i = 0; i < points.length; i++) {  Vector v = new RandomAccessSparseVector(points[i].length);  v.assign(points[i]);  vList.add(v);  }  return vList;  }    public static void main(String args[]) throws Exception {  List<Vector> pointsList = createVectors(points);  List<Canopy> canopiesList = CanopyClusterer.createCanopies(  pointsList, new EuclideanDistanceMeasure(), 8.0, 4.0);  for (Canopy canopy : canopiesList) {  System.out.println(canopy.getId() + " : " + canopy.getCenter());  }  }  } |

実行例を以下に示します。

k平均法クラスタリングと同様、3つのクラスタの重心が得られました。

|  |
| --- |
| $ mahout CanopyClustererSample  ...  0 : {0:1.0,1:1.0}  1 : {0:1.6666666666666667,1:9.333333333333334}  2 : {0:10.0,1:6.0} |

### CanopyDriverを利用した実装例

Mahoutによるキャノピークラスタリングの別の実装例を示します。

ここでは、CanopyDriverクラスを使用します。KMeansDriverと同様、runメソッドで実行します。引数に、入力データ(シーケンスファイル)、出力ディレクトリ、距離測度、、、クラスタリングの実施(true)などです。キャノピークラスタリングでは、クラスタの重心の初期値を必要としませんが、それを除けば、他の処理は基本的に同様です。

|  |
| --- |
| CanopyDriver.run(conf, new Path("cpinput" + File.separator + "points"),  new Path("cpoutput"), new EuclideanDistanceMeasure(), 8.0, 4.0, true, 0.0, true); |

プログラムの全体を以下に示します。

|  |
| --- |
| import org.apache.hadoop.conf.\*;  import org.apache.hadoop.fs.\*;  import org.apache.hadoop.io.\*;  import org.apache.mahout.clustering.classify.\*;  import org.apache.mahout.clustering.canopy.\*;  import org.apache.mahout.clustering.iterator.\*;  import org.apache.mahout.common.distance.\*;  import org.apache.mahout.math.\*;  import java.io.\*;  import java.util.ArrayList;  import java.util.List;  public class CanopySample {  public static final String OUTPUTDIR = "cpoutput";  public static final String BASEDIR = "cpinput";  public static final String POINTSDIR = BASEDIR + File.separator + "points";  public static final double[][] points = {  {0, 1}, {1, 1}, {2, 1}, {1, 9}, {2, 9}, {2, 10},  {9, 5}, {10, 6}, {11, 6}, {10, 7},  };    public static List<Vector> createVectors(double[][] points) {  List<Vector> vList = new ArrayList<Vector>();  for (int i = 0; i < points.length; i++) {  Vector v = new RandomAccessSparseVector(points[i].length);  v.assign(points[i]);  vList.add(v);  }  return vList;  }    public static void main(String args[]) throws Exception {  List<Vector> pointsList = createVectors(points);  File dir = new File(BASEDIR);  if (! dir.exists()) {  dir.mkdir();  }  dir = new File(POINTSDIR);  if (! dir.exists()) {  dir.mkdir();  }  Configuration conf = new Configuration();  FileSystem fs = FileSystem.get(conf);  SequenceFile.Writer writer = new SequenceFile.Writer(  fs, conf, new Path(POINTSDIR + File.separator + "file1"),  LongWritable.class, VectorWritable.class);  VectorWritable vw = new VectorWritable();  long num = 0;  for (Vector point : pointsList) {  vw.set(point);  writer.append(new LongWritable(num++), vw);  }  writer.close();    CanopyDriver.run(conf, new Path(POINTSDIR), new Path(OUTPUTDIR),  new EuclideanDistanceMeasure(), 8.0, 4.0, true, 0.0, true);  SequenceFile.Reader reader = new SequenceFile.Reader(fs,  new Path(OUTPUTDIR + File.separator + Canopy.CLUSTERS\_DIR + "0" +  Canopy.FINAL\_ITERATION\_SUFFIX + File.separator +  "part-r-00000"), conf);  Text key1 = new Text();  ClusterWritable value1 = new ClusterWritable();  while (reader.next(key1, value1)) {  System.out.println(value1.getValue());  }  reader.close();  reader = new SequenceFile.Reader(fs,  new Path(OUTPUTDIR + File.separator + Canopy.CLUSTERED\_POINTS\_DIR +  File.separator + "part-m-0"), conf);  IntWritable key2 = new IntWritable();  WeightedVectorWritable value2 = new WeightedVectorWritable();  while (reader.next(key2, value2)) {  System.out.println(key2 + " : " + value2);  }  reader.close();  }  } |

実行例を以下に示します。

4.4.2と同様、3つのクラスタの重心が得られました。また、10個の点が4.3.2と同様にクラスタへ割り振られていることがわかります。

|  |
| --- |
| $ mahout CanopySample  ...  C-0: {0:1.0,1:1.0}  C-1: {0:1.6666666666666667,1:9.333333333333334}  C-2: {0:10.0,1:6.0}  0 : 0.7273988629009037: [1:1.000]  0 : 0.8365625322160762: [1.000, 1.000]  0 : 0.7113315478811213: [2.000, 1.000]  1 : 0.7350987050339858: [1.000, 9.000]  1 : 0.7595685946297487: [2.000, 9.000]  1 : 0.7412415128365745: [2.000, 10.000]  2 : 0.6682604045237258: [9.000, 5.000]  2 : 0.8412001835161694: [10.000, 6.000]  2 : 0.7421122102145497: [11.000, 6.000]  2 : 0.7265202263032431: [10.000, 7.000] |

### mahoutコマンドによる実行例

mahoutコマンドでも、キャノピークラスタリングを実行することが可能です。

実行例を以下に示します。

|  |
| --- |
| $ mahout canopy --input cpinput/points --output cpoutput2 \  --distanceMeasure org.apache.mahout.common.distance.EuclideanDistanceMeasure \  --t1 8 --t2 4 |

キャノピークラスタリングを行うためのコマンド名は「canopy」です。--inputオプションでは4.4.3で作成したシーケンスファイルを指定します。また、--distanceMeasureで距離測度、--t1および--t2でおよびを指定します。。

上記を実行すると、4.4.3と同様、シーケンスファイルが出力として得られます。これは、クラスタの重心の初期値が格納されたシーケンスファイルです。これをテキストに変換するには、mahoutの「clusterdump」コマンドを用います。

|  |
| --- |
| $ mahout clusterdump --input cpoutput2/clusters-\*-final  ...  C-0{n=1 c=[1.000, 1.000] r=[]}  C-1{n=1 c=[1.667, 9.333] r=[]}  C-2{n=1 c=[10.000, 6.000] r=[]} |

**演習問題**

ここでは、4.4.3で示した、CanopyDriverクラスを利用した実装例をもとに、「フィッシャーのアヤメ」という有名なデータに対して、キャノピークラスタリングを行います。

フィッシャーのアヤメのデータは、アヤメの標本150個のがく片の長さと幅、花弁の長さと幅の測定値から構成されます。具体的には、下記のような内容になっています。

|  |
| --- |
| $ cat iris.txt  Sepal.Length Sepal.Width Petal.Length Petal.Width Species  1 5.1 3.5 1.4 0.2 setosa  2 4.9 3 1.4 0.2 setosa  3 4.7 3.2 1.3 0.2 setosa  ...  51 7 3.2 4.7 1.4 versicolor  52 6.4 3.2 4.5 1.5 versicolor  53 6.9 3.1 4.9 1.5 versicolor  ...  148 6.5 3 5.2 2 virginica  149 6.2 3.4 5.4 2.3 virginica  150 5.9 3 5.1 1.8 virginica |

キャノピークラスタリングを行うデータを、ファイルから入力したいので、以下のクラスをインポートします。

|  |
| --- |
| import java.io.\*;  import java.util.ArrayList;  import java.util.List;  +import java.io.FileReader;  +import java.io.BufferedReader; |

入力するファイルはCSV形式とします。新たに、ファイル名を引数に受け取りVectorクラスのリストを返す関数を追加します。

|  |
| --- |
| }  return vList;  }  +  + public static List<Vector> convertVectors(String filename) throws Exception  + {  + List<Vector> vList = new ArrayList<Vector>();  + BufferedReader reader = new BufferedReader(new FileReader(filename));  + String line;  + while ((line = reader.readLine()) != null) {  + String[] c = line.split(",");  + double[] d = new double[c.length];  + for (int i = 0; i < c.length; i++) {  + d[i] = Double.parseDouble(c[i]);  + }  + Vector v = new RandomAccessSparseVector(c.length);  + v.assign(d);  + vList.add(v);  + }  + reader.close();  + return vList;  + } |

コマンドラインから、ファイル名を受け取るように変更します。

|  |
| --- |
| public static void main(String args[]) throws Exception {  - List<Vector> pointsList = createVectors(points);  + List<Vector> pointsList = convertVectors(args[0]);    File dir = new File(BASEDIR);  if (! dir.exists()) { |

同様に、コマンドラインから、閾値およびを受け取るように変更します。

|  |
| --- |
| writer.close();    CanopyDriver.run(conf, new Path(POINTSDIR), new Path(OUTPUTDIR),  - new EuclideanDistanceMeasure(), 8.0, 4.0, true, 0.0, true);  + new EuclideanDistanceMeasure(), Double.parseDouble(args[1]),  + Double.parseDouble(args[2]), true, 0.0, true);    SequenceFile.Reader reader = new SequenceFile.Reader(fs,  new Path(OUTPUTDIR + File.separator + Canopy.CLUSTERS\_DIR + "0" + |

キャノピークラスタリングを行った結果から、ベクトルのみ表示するように変更します。

|  |
| --- |
| IntWritable key2 = new IntWritable();  WeightedVectorWritable value2 = new WeightedVectorWritable();  while (reader.next(key2, value2)) {  - System.out.println(key2 + " : " + value2);  + System.out.println(key2 + " : " + value2.getVector());  }  reader.close();  } |

上記をコンパイルおよび実行するためには、クラスパスを設定する必要があります。

|  |
| --- |
| $ export HADOOP\_LIB\_DIR=/usr/lib/hadoop  $ export MAHOUT\_HOME=/usr/local/mahout  $ CLASSPATH=.  $ CLASSPATH=${HADOOP\_LIB\_DIR}/hadoop-tools.jar:${CLASSPATH}  $ CLASSPATH=${HADOOP\_LIB\_DIR}/hadoop-ant.jar:${CLASSPATH}  $ CLASSPATH=${HADOOP\_LIB\_DIR}/hadoop-core.jar:${CLASSPATH}  $ CLASSPATH=${MAHOUT\_HOME}/mahout-math-0.8.jar:${CLASSPATH}  $ CLASSPATH=${MAHOUT\_HOME}/mahout-integration-0.8.jar:${CLASSPATH}  $ CLASSPATH=${MAHOUT\_HOME}/mahout-examples-0.8-job.jar:${CLASSPATH}  $ CLASSPATH=${MAHOUT\_HOME}/mahout-examples-0.8.jar:${CLASSPATH}  $ CLASSPATH=${MAHOUT\_HOME}/mahout-core-0.8-job.jar:${CLASSPATH}  $ CLASSPATH=${MAHOUT\_HOME}/mahout-core-0.8.jar:${CLASSPATH}  $ CLASSPATH=/usr/share/java/log4j.jar:${CLASSPATH}  $ export CLASSPATH |

コンパイルは、以下のように行います。

|  |
| --- |
| $ javac -classpath ${CLASSPATH} CanopySample.java |

今回は花弁の長さと幅を使いたいので、下記のように、CSV形式に変換します。

|  |
| --- |
| $ awk '/^[1-9]/ {printf "%s,%s\n", $4,$5}' iris.txt > petal.csv |

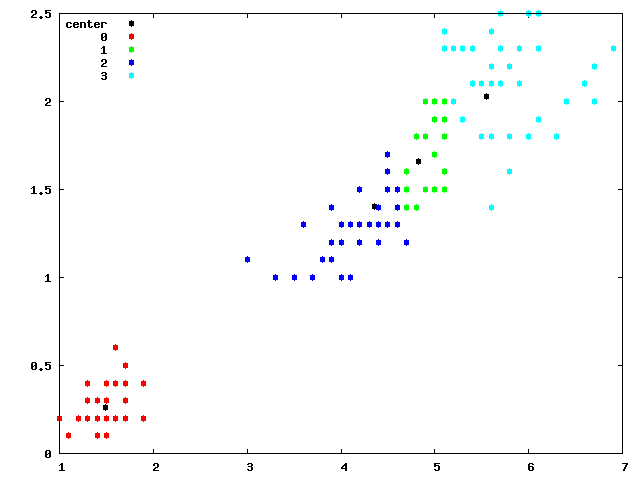
実行例を以下に示します。

閾値を2.0、を1.0とした場合、4つのクラスタの重心が得られました。

|  |
| --- |
| $ java -classpath ${CLASSPATH} CanopySample petal.csv 2.0 1.0  …  C-0: {0:1.4921568627450983,1:0.2627450980392156}  C-1: {0:4.830208333333335,1:1.6562500000000009}  C-2: {0:4.3537037037037045,1:1.4037037037037037}  C-3: {0:5.552,1:2.026}  0 : {0:1.4,1:0.2}  0 : {0:1.4,1:0.2}  0 : {0:1.3,1:0.2}  0 : {0:1.5,1:0.2}  … |

プロットすると、以下のようになります。

centerはクラスタの重心です。0～3は各クラスタのベクトルです。



**図 1花弁の長さと幅でキャノピークラスタリング()**

プロットには、「gnuplot」コマンドを使用します。

下記に、0～11までのクラスタをプロットするシェルスクリプトを示します。

|  |
| --- |
| #!/bin/sh  grep "^C" ${1} | awk -F [:\{,\}] '{printf "%s,%s\n", $4,$6}' > center.csv  if [ ! -d cluster ]; then  mkdir cluster  fi  for i in `seq 0 11`; do  grep "^${i}" ${1} | awk -F [:\{,\}] '{printf "%s,%s\n", $4,$6}' > cluster/${i}.csv  done  gnuplot << EOF 1> /dev/null 2> /dev/null  set datafile separator ","  set key left  set terminal png  plot "center.csv" pt 7 lt rgb "#000000" title "center"  replot "cluster/0.csv" pt 7 lt rgb "#FF0000" title "0"  replot "cluster/1.csv" pt 7 lt rgb "#00FF00" title "1"  replot "cluster/2.csv" pt 7 lt rgb "#0000FF" title "2"  replot "cluster/3.csv" pt 7 lt rgb "#00FFFF" title "3"  replot "cluster/4.csv" pt 7 lt rgb "#FF00FF" title "4"  replot "cluster/5.csv" pt 7 lt rgb "#FFFF00" title "5"  replot "cluster/6.csv" pt 7 lt rgb "#800000" title "6"  replot "cluster/7.csv" pt 7 lt rgb "#008000" title "7"  replot "cluster/8.csv" pt 7 lt rgb "#000080" title "8"  replot "cluster/9.csv" pt 7 lt rgb "#008080" title "9"  replot "cluster/10.csv" pt 7 lt rgb "#800080" title "10"  replot "cluster/11.csv" pt 7 lt rgb "#808000" title "11"  set output "result.png"  replot  EOF |

実行例を以下に示します。

result.pngという名前のファイルにプロットされます。

|  |
| --- |
| $ ./graph-from-canopy.sh result.txt  $ ls  center.csv cluster graph-from-canopy.sh result.png result.txt |

上記の変更をCanopySampleクラスに対して行い、閾値およびを少しずつ変更して、クラスタの重心の変化を見てみましょう。

# 分類

## 概要

分類とは、特定の情報(入力)を用いて、あらかじめ決められた取りうる選択肢の中から、単一の回答(出力)を選ぶ処理のことです。たとえば、スパムフィルタの場合、メールを入力として与えると、スパムかもしくはスパムでないか、いずれかを回答として出力します。手書き文字認識の場合は、手で書いた文字を画像として入力すると、文字を出力します。

4章で解説したクラスタリングでは、回答の候補となる選択肢が決まっておらず、アルゴリズムによって回答となるクラスタそのものを推定します。このように、入力だけ与えられ、アルゴリズム自体が回答を見つける必要のある学習を、「教師なし学習」と言います。これに対して分類では、入力と正しい結果のペアを用いて学習を行い、新たな入力データの回答を推定します。このように、入力と期待される出力を用いて行う学習を「教師あり学習」と言います。

分類の動作は、学習ステップと分類ステップの2つに分けられます(図 7)。学習ステップでは、「学習アルゴリズム」に従って、回答を選択する「モデル(分類モデル)」の学習を行います。まず、入力と出力のペアで与えられる「学習サンプル」を、「学習データ」と「評価データ」に分けます。そして、学習データを用いてモデルの学習を行います。次に、評価データを用いて、学習したモデルの評価を行います。

分類ステップでは、新たなデータをモデルに入力して、実際に分類を行います。

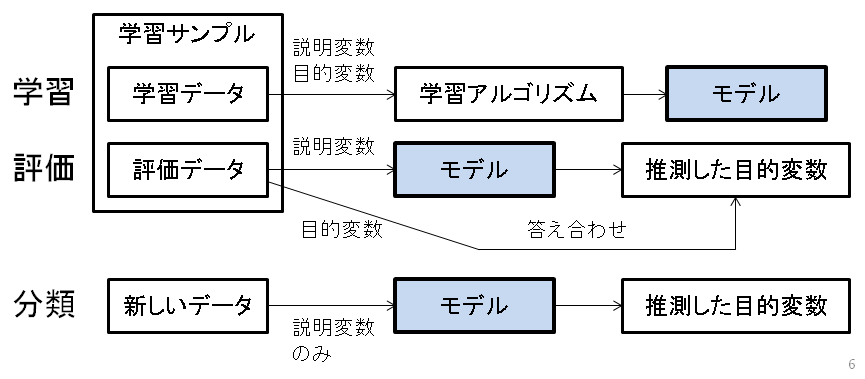


図 8分類の動作

分類では、モデルに与える入力を「説明変数」、モデルが推定しようとする出力を「目的変数」と呼びます。

以降では、分類の代表的なアルゴリズムである「単純ベイズ法」と、「決定木」および「Random Forest」について解説します。また、モデルの出力を評価する方法についても述べます。

## 単純ベイズ法

### 概要

単純ベイズ法(ナイーブベイズ法)では、「ベイズの公式(定理)」に対して、説明変数が互いに独立であるという仮定を適用することで分類を行います。

確率変数XとYの同時確率p(X,Y)は、条件付き確率p(X|Y)にp(Y)をかけたもの、あるいはp(Y|X)にp(X)をかけたものです。

これをp(Y)で割ると、ベイズの公式が得られます。

ここで、Xが目的変数、が説明変数とすると、説明変数が与えられたときの目的変数の条件付き確率 p(X|,,…) は、以下の式で表されます。

学習データから、目的変数の確率p(X)、説明変数の確率p(,,…)は求められますが、説明変数が与えられたときの目的変数の事後確率 p(,,…|X)を求めることができません。そのため、単純ベイズ法では、説明変数同士が互いに独立と仮定した推定値を用います。確率変数AとBが独立の場合、以下の式が成り立ちます。

これを前述の式に当てはめると、下記の式が得られます。

これにより、説明変数が与えられたときの目的変数の条件付き確率を、それぞれ求めることができます。その中から、もっとも高い確率の目的変数を、分類の結果として出力します。

### Mahoutによる単純ベイズ法の実行例

Mahoutによる単純ベイズ法の分類の実行例を示します。スパムメールの検出を行います。大まかな処理の流れは以下の通りです。

1. 形態素解析ソフトで分かち書きしたメールの学習データおよび評価データを用意する。
2. それぞれのデータをベクトルファイルに変換する。
3. 学習データを用いて、モデルを生成する(学習する)。
4. 評価データを用いて、生成したモデルを評価する。
5. 新たなデータを、モデルによって分類する。

1.に関しては、すでに分かち書きされたメールがあると仮定します。学習データ用のディレクトリ(mail-train)と、評価データ用のディレクトリ(mail-test)に、スパムメールを格納するディレクトリ(spam)とそうでないメールを格納するディレクトリ(ham)を作成し、それぞれにデータとなるメールを置きます。例を以下に示します。

|  |
| --- |
| $ tree mail-train  mail-train  |-- ham  | |-- ham1.txt  | |-- ham2.txt  ...  |-- spam  | |-- spam1.txt  | |-- spam2.txt  ... |

2.では、mahoutコマンドを使用して、シーケンスファイルに一旦変換した後、ベクトルファイルに変換します。ディレクトリ以下にあるテキストファイルをシーケンスファイルに変換するには、「seqdirectory」コマンドを使用します。

|  |
| --- |
| $ mahout seqdirectory --input mail-train --output mail-train-seq  $ mahout seqdirectory --input mail-test --output mail-test-seq |

生成されたシーケンスファイルは、ファイル名をkey、本文をvalueとしています。

このシーケンスファイルをベクトルファイルに変換するには、「seq2sparse」コマンドを使用します。

|  |
| --- |
| $ mahout seq2sparse --input mail-train-seq --output mail-train-vec  $ mahout seq2sparse --input mail-test-seq --output mail-test-vec |

生成されたベクトルファイルは、本文中に出現する単語を次元とし、出現頻度TF(Term Frequency)や、TF-IDF(Term Frequency-Inverse TF)の重みを値としたものです。それぞれ、以下の式で表されます。

ここで、は単語iが文書jに出現する回数、は総文書数、は単語iを含む文書数です。TFは、TF-IDFはにをかけたものを使用します。TF-IDFでは、をかけることで、一般的な単語の重要度を下げ、低頻度の単語の重要度を上げています。

--outputオプションで指定したディレクトリの中には、TF(tf-vectors)やTF-IDF(tfidf-vectors)、辞書ファイル(dictionary.file-0)などが含まれます。いずれも、前述のseqdumperコマンドで内容を確認できます。

3.では、「trainnb」コマンドで、単純ベイズ法による学習を行います。

|  |
| --- |
| $ mahout trainnb --input mail-train-vec/tfidf-vectors \  --output mail-nbmodel --extractLabels --labelIndex mail-labelindex |

学習した結果、モデルを出力します。ここでは、TF-IDFのベクトルファイルを入力に用いています。また、「--extractLabels」オプションでラベルファイルの出力を、「--labelIndex｣オプションでそのファイル名を指定しています。これもシーケンスファイルです。

|  |
| --- |
| $ mahout seqdumper --input mail-labelindex  ...  Key: ham: Value: 0  Key: spam: Value: 1 |

スパムの場合は1、スパムでない場合は0となっています。

4.では、「testnb」コマンドで評価を行います。

|  |
| --- |
| $ mahout testnb --input mail-test-vec/tfidf-vectors --output mail-testnb-out \  --model mail-nbmodel --labelIndex mail-labelindex |

評価データを入力し、「--model」オプションで生成したモデル、「--labelIndex」オプションでラベルファイルを指定します。

実行すると、評価結果を出力します。

|  |
| --- |
| ======================================================  Summary  -------------------------------------------------------  Correctly Classified Instances : 10 55.5556%  Incorrectly Classified Instances : 8 44.4444%  Total Classified Instances : 18  =======================================================  Confusion Matrix  -------------------------------------------------------  a b <--Classified as  7 1 | 8 a = ham  7 3 | 10 b = spam  =======================================================  Statistics  -------------------------------------------------------  Kappa 0.2048  Accuracy 55.5556%  Reliability 39.1667%  Reliability (standard deviation) 0.4446 |

正解率は約56%でした。あまり精度はよくありませんが、データがもっと大量にあれば、もう少し結果がよくなるはずです。

5.の分類ですが、mahoutコマンドには該当するものがないため、Javaで実装します。

まず、NaiveBayesModelクラスを使用してモデルファイルを読み込みます。

|  |
| --- |
| Configuration conf = new Configuration();  NaiveBayesModel model = NaiveBayesModel.materialize("mail-nbmodel", conf); |

次に、分類を行うStandardNaiveBayesClassifierクラスのインスタンスを生成します。

|  |
| --- |
| StandardNaiveBayesClassifier classifier = new StandardNaiveBayesClassifier(model); |

辞書ファイルの読み込みには、SequenceFileIteratorクラスを使用します。

|  |
| --- |
| SequenceFileIterator<?, ?> dictIterator =  new SequenceFileIterator<Writable, Writable>(new Path("dictionary.file-0"), true, conf); |

これは、単語の出現回数をカウントする際に利用するため、HashMapに変換しておきます。

入力ファイル(分類対象のテキストファイル)は、読み込んだ後、スペースで区切られた単語のリストに変換して、さらに単語IDとその出現回数からなるVectorへ変換します。

|  |
| --- |
| Vector vector = new RandomAccessSparseVector(dictMap.size()); // dictMapは辞書  for (String key : loadFile(args[0])) {  if(dictMap.get(key) != null) {  int value = dictMap.get(key);  vector.setQuick(value, vector.get(value)+1); // 出現回数をインクリメント  }  } |

そして、モデルを使用して分類を行います。

|  |
| --- |
| Vector result = classifier.classifyFull(vector); |

分類を行うプログラムの全体を、以下に示します。

|  |
| --- |
| import org.apache.hadoop.conf.\*;  import org.apache.hadoop.fs.\*;  import org.apache.hadoop.io.\*;  import org.apache.mahout.classifier.naivebayes.\*;  import org.apache.mahout.common.\*;  import org.apache.mahout.common.iterator.sequencefile.\*;  import org.apache.mahout.math.\*;  import java.io.\*;  import java.util.ArrayList;  import java.util.Arrays;  import java.util.HashMap;  import java.util.List;  public class NBSample {  public static List<String> loadFile(String path) throws Exception {  BufferedReader reader = new BufferedReader(  new FileReader(new File(path)));  ArrayList<String> list = new ArrayList<String>();  String str;  while ((str = reader.readLine()) != null) {  list.addAll(Arrays.asList(str.split("[ ]+")));  }  reader.close();  return list;  }  public static void main(String args[]) throws Exception {  if (args.length != 3) {  System.err.println("Usage: java NBSample input\_file dicti\_file model\_dir");  System.exit(1);  }  Configuration conf = new Configuration();  NaiveBayesModel model =  NaiveBayesModel.materialize(new Path(args[2]), conf);  StandardNaiveBayesClassifier classifier =  new StandardNaiveBayesClassifier(model);  SequenceFileIterator<?, ?> dictIterator =  new SequenceFileIterator<Writable, Writable>(  new Path(args[1]), true, conf);  HashMap<String, Integer> dictMap = new HashMap<String, Integer>();  while (dictIterator.hasNext()) {  Pair<?, ?> dict = dictIterator.next();  String key = dict.getFirst().toString();  Integer value = Integer.valueOf(dict.getSecond().toString());  dictMap.put(key, value);  }  Vector vector = new RandomAccessSparseVector(dictMap.size());  for (String key : loadFile(args[0])) {  if(dictMap.get(key) != null) {  int value = dictMap.get(key);  vector.setQuick(value, vector.get(value)+1);  }  }  Vector result = classifier.classifyFull(vector);  for (int i = 0; i < result.size(); i++) {  System.out.println(i + " : " + result.get(i));  }  }  } |

実行例を以下に示します。1つ目がスパムメール、2つ目がスパムでないメールを分類したときの実行例です。いずれも、僅差ではありますが、正しく分類されています。

|  |
| --- |
| $ mahout NBSample mail-data/spam/spam1.txt \  mail-train-vec/dictionary.file-0 mail-nbmodel  ...  0 : -2453.8582528860907  1 : -2392.390795088545  $ mahout NBSample mail-data/ham/ham1.txt \  mail-train-vec/dictionary.file-0 mail-nbmodel  ...  0 : -9207.733818332366  1 : -9289.136598953224 |

## 決定木

### 概要

決定木では、説明変数を使った単純なルールで分岐を作り、データを分割して分類を行います。5.2節の単純ベイズ法では、各説明変数の重要度は(確率の値から)わかるものの、どのような理由で目的変数を推定したか、その過程を知ることができません。決定木の場合、図 8に示すように、ルートノードから順にルールを適用することで目的変数を推定するため、推論過程を確認することができます。

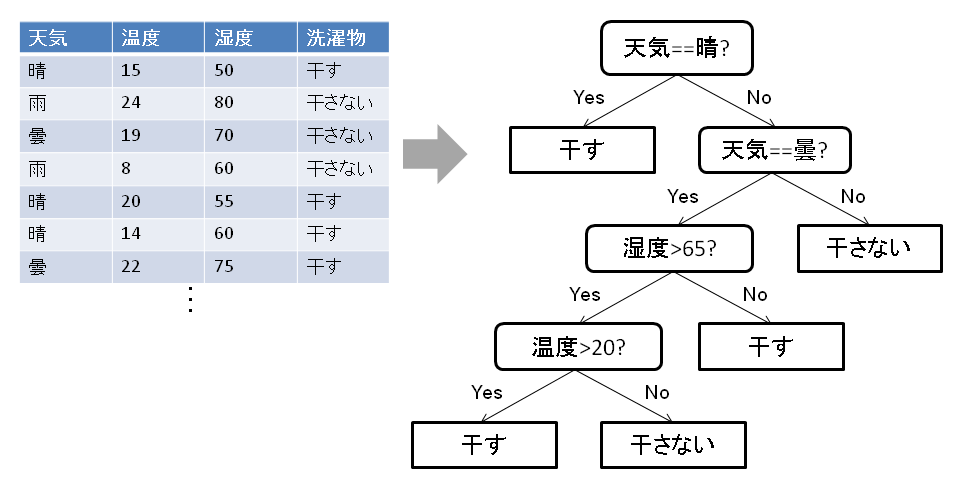


図 9決定木の例

決定木の生成には、「CART(Classification And Regression Trees)」というアルゴリズムを使用します。アルゴリズムは以下の通りです。

1. ノードを生成する。
2. データをもっともよく分割する(純化する)説明変数を選び、分割条件をそのノードのルールとする。具体的には、データAをとに分割するとき、以下の式が最大になるように分割する。ここで、I(A)は不純度を表す。
3. 分割されたデータに対して、取りうる目的変数が2つ以上のとき、1.～2.を繰り返す。

1つの場合、目的変数の値のリーフノードを生成する。

不純度には、主に「GINI係数」と「エントロピー」が使用されます。

### GINI係数

GINI変数は、誤差率(推測が正しくない率)を表す不純度です。と推測していたにもかかわらず、実際にはが発生してしまったときの確率です。以下の式で表されます。0から1の間の値をとります。

### エントロピー

エントロピーは、集合中の無秩序の量を表します。以下の式で表されます。同士がかけ離れているほど大きい値になります。

= 1 -

## Random Forest

### 概要

Random Forestは、複数の決定木を用いて、多数決で出力を決定する分類のアルゴリズムです(図 9)。高速に処理できる、並列処理しやすい、ノイズに強いなどの利点があります。アルゴリズムは以下の通りです。

1. 学習データをもとに、B組のサンプルをランダムに生成する。
2. 1.で生成したサブサンプルごとに、決定木を作成する。
3. 分類の際には、各決定木ごとに推定を行い、多数決で出力を決定する。

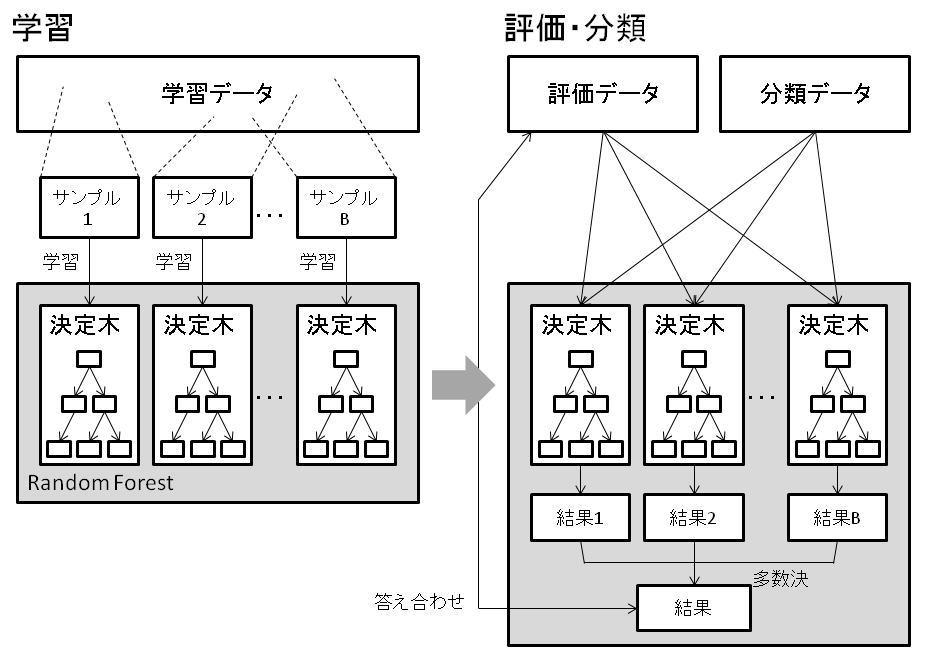


図 10Random Forestの動作

Mahoutでは、Random Forestではなく、「Decision Forest」と呼んでいます。

### mahoutコマンドによる実行例

フィッシャーのアヤメのデータを使用した、Random Forestによる分類の実行例を示します。

mahoutコマンドでRandom Forestによる分類を行う場合、入力をCSVファイルにする必要があります。たとえば下記のように実行して、CSV形式に変換します。

|  |
| --- |
| $ awk '/^[1-9]/ {printf "%s,%s,%s,%s,%s\n", $2,$3,$4,$5,$6}' iris.txt > iris.csv |

学習データと評価データに分け、HDFSに置きます。

|  |
| --- |
| $ awk 'NR%3!=0' iris.csv > iris-train.csv  $ awk 'NR%3==0' iris.csv > iris-test.csv  $ hadoop fs -put iris-train.csv mahout-data  $ hadoop fs -put iris-test.csv mahout-data |

次に、CSVファイルの各データを説明するためのディスクリプタファイルを、Describeクラスを用いて作成します。

|  |
| --- |
| $ mahout org.apache.mahout.classifier.df.tools.Describe \  --path mahout-data/iris-train.csv --file mahout-data/iris-dataset \  --descriptor 4 N L |

「--path」オプションで入力ファイル、「--file」オプションで出力となるディスクリプタファイルを指定します。「--descriptor」オプションでは、入力ファイルの各カラムの形式を指定します。「N」は数値の説明変数、「C」はカテゴリの説明変数、「I」は無視(使用しない)、「L」は目的変数を示します。また、整数の場合は、直後の文字をその数だけ繰り返します。たとえば、「N N N N」は「4 N」と表せます。

そして、BuildForestクラスを使用して学習を行います。

|  |
| --- |
| $ mahout org.apache.mahout.classifier.df.mapreduce.BuildForest \  --data mahout-data/iris-train.csv --dataset mahout-data/iris-dataset \  --output mahout-data/iris-model-df --nbtrees 3 |

「--data」オプションで学習データ、「--dataset」オプションでディスクリプタファイル、「--output」オプションで出力となるモデルを指定します。また、「--nbtree」オプションで、決定木の数を指定します。

モデルが生成された後は、評価を行います。TestForestクラスを用います。評価データのCSVファイルを入力とし、--datasetオプションでディスクリプタファイル、「--model」オプションでモデルを指定しています。また、「--analyze」オプションを指定すると、下記のように結果を出力します。「--mapreduce」オプションを指定すると、Hadoopによる分散処理を行います。

|  |
| --- |
| $ mahout org.apache.mahout.classifier.df.mapreduce.TestForest \  --input mahout-data/iris-test.csv --dataset mahout-data/iris-dataset \  --model mahout-data/iris-model-df --analyze --mapreduce \  --output mahout-data/iris-test-output  ...  =======================================================  Summary  -------------------------------------------------------  Correctly Classified Instances : 47 94%  Incorrectly Classified Instances : 3 6%  Total Classified Instances : 50  =======================================================  Confusion Matrix  -------------------------------------------------------  a b c <--Classified as  16 1 0 | 17 a = virginica  2 15 0 | 17 b = versicolor  0 0 16 | 16 c = setosa  =======================================================  Statistics  -------------------------------------------------------  Kappa 0.7992  Accuracy 94%  Reliability 70.5882%  Reliability (standard deviation) 0.473 |

## 評価

分類モデルの評価を行うための性能指標について、代表的なものをいくつか紹介します。

### 正解率

全評価に対する正解数の割合です。正解数しか見ないため、一意の値しか出力しないモデルの方が、正解に近い結果を出力するモデルよりも高評価になることがあります。

### 混同行列

目的変数の実際の正しい結果を行、モデルの出力を列とした集計表です。対角線上の数値が、正しく分類された結果の数です。正しさの度合いが一目でわかります。

5.4.2の評価の結果に混同行列が含まれています。

|  |
| --- |
| =======================================================  Confusion Matrix  -------------------------------------------------------  a b c <--Classified as  16 1 0 | 17 a = virginica  2 15 0 | 17 b = versicolor  0 0 16 | 16 c = setosa |

### エントロピー行列

混合行列の行列内の値をエントロピーで表した行列です。

### AUC

AUC(Area Under the Curve)では、目的変数が0,1の値をとるとき、実際の値が1の場合の正解率「感度」と、実際の値が0の場合の正解率「特異度」の2つで評価します。

推定が1で実際の値も1の個数をTP、推定が1で実際の値が0の個数をFP、推定が0で実際の値も0の個数をTN、推定が0で実際の値が1の個数をFNとすると、それぞれ以下の式で表されます。

|  |  |  |  |
| --- | --- | --- | --- |
|  | | 実際 | |
| 1 | 0 |
| 出力 | 1 | TP | FP |
| 0 | FN | TN |

一般的には、0.7～0.9の場合よいとされます。ランダムの場合は 0.5 です。

**演習問題**

下記に、コマンドラインから数を指定し、その数だけ各アヤメの学習データを抽出するシェルスクリプトを示します。

|  |
| --- |
| #!/bin/sh  grep 'setosa' iris.csv | head -n ${1} > iris-train.csv  grep 'versicolor' iris.csv | head -n ${1} >> iris-train.csv  grep 'virginica' iris.csv | head -n ${1} >> iris-train.csv |

実行例を以下に示します。

指定した数だけ、各アヤメの学習データがiris-train.csvという名前のファイルに、抽出されています。

|  |
| --- |
| $ ./generate-iris-train.sh 1  $ ls  generate-iris-train.sh iris.csv iris-train.csv  $ cat iris-train.csv  5.1,3.5,1.4,0.2,setosa  7,3.2,4.7,1.4,versicolor  6.3,3.3,6,2.5,virginica |

上記のシェルスクリプトを使って、学習データを少しずつ増やしながら、正解率、混同行列の変化を見てみましょう。

# 参考文献

* Sean Owen, Robin Anil, Ted Dunning, Ellen Friedman 著, 伊東直子, 真鍋加奈子, 堀内孝彦, 都元ダイスケ 訳: 「Mahoutイン・アクション」オライリー・ジャパン, 2012
* Toby Segaran 著, 當山仁健, 鴨澤眞夫 訳: 「集合知プログラミング」オライリー・ジャパン, 2008
* 太田一樹, 岩崎正剛, 猿田浩輔, 下垣徹, 藤井達朗, 山下真一:「Hadoop徹底入門 第2版」翔泳社, 2013
* 中野猛, 山下真一, 猿田浩輔, 上新卓也, 小林隆:「Hadoop Hacks－プロフェッショナルが使う実践テクニック」オライリー・ジャパン, 2012.

# 変更履歴

|  |  |  |
| --- | --- | --- |
| **版数** | **変更内容** | **年月日** |
| 1.0 | 新規作成 | 2014/MAR/3 |