Apache Mahout の紹介

インテリジェントなアプリケーションを作成するための、スケーラブルで商用化しやすい機械学習

Grant Ingersoll
Member, Technical Staff
Lucid Imagination

2009年 9月 08日

データやユーザー入力から学習するインテリジェントなアプリケーションの開発は、これまでは巨額の研究予算のある学術界や企業でしか行われていませんでしたが、今や一般的になりつつあります。クラスタリングや協調フィルタリング、カテゴリー分けなど、機械学習の手法に対するニーズは今までにないほど高まっています。機械学習の用途には、大勢の人達の間での共通性の発見、あるいは大量の Web コンテンツの自動タグ付けなどがあります。Apache Mahout プロジェクトは、インテリジェントなアプリケーションの作成を容易に、そして迅速にすることを目標としています。この記事では Mahout プロジェクトの共同設立者である Grant Ingersoll が機械学習の基本的な概念を紹介し、Mahout を使って文書をクラスタリングする方法、レコメンデーションを行う方法、コンテンツを構成する方法を説明します。

情報化時代における企業や個人の成功は、膨大な量のデータを、いかに速く効率的に実用的な情報に変えられるかに左右される傾向が強まりつつあります。毎日何百通あるいは何千通という個人的な E メール・メッセージを処理する場合であれ、ペタバイトのブログからユーザーの意図を探り当てる場合であれ、データを整理し、強化することができるツールへのニーズが、これまでなかったほど高まっています。そこに、機械学習の分野、そしてこの記事で紹介する Apache Mahout プロジェクト (「参考文献」を参照) の前提と展望があります。

機械学習は人工知能の副分野であり、コンピューターが過去の経験に基づいて出力を改善するための手法に焦点を当てます。この分野はデータ・マイニングと密接に関連しており、また多くの場合は統計、確率論、パターン認識、その他多くのさまざまな分野の手法を利用します。機械学習は新しい分野ではありませんが、明らかに成長を続けています。IBM®、Google、Amazon、Yahoo!、Facebook など、多くの大企業が彼らのアプリケーションに機械学習のアルゴリズムを実装しています。こうした企業以外のさらに多くの企業にとっても、ユーザーや過去の状況から学ぶために企業アプリケーションに機械学習を活用することには大きなメリットがあります。

この記事では機械学習の概念を簡単に説明した後、Apache Mahout プロジェクトの歴史、目標、そして機能を紹介します。次に Mahout を使用して、無料で利用できるウィキペディアのデータ・セットを使用する興味深い機械学習タスクについて説明します。

機械学習の基礎

機械学習では、ゲームのプレイ、不正の検出、そして株式市場の分析まで、あらゆるものを使用します。機械学習は、Netflix や Amazon などのサイトで過去の購買履歴に基づいてユーザーに商品を推薦するシステムの構築や、指定された日付における類似のニュース記事をすべて見つけるシステムの構築などに使われています。また機械学習は、Web ページをジャンル (スポーツ、経済、戦争、等々) に従って自動的にカテゴリー分けするためや、E メール・メッセージからスパム・メールを検出するためにも使われています。機械学習の用途は、この記事で説明しきれないほど数多くあります。この分野について詳しく知りたい方は「参考文献」を参照してください。

機械学習では、いくつかの手法を使って問題を解決します。ここでは最も一般的に使われる 2 つの手法、教師あり学習と教師なし学習に焦点を絞ります。その理由は、この 2 つが Mahout でサポートされている中心的な手法だからです。

教師あり学習では、ラベル付きの訓練データから機能を学習し、有効な任意の入力の値を予測します。教師あり学習の一般的な例として、E メール・メッセージからスパム・メールを分類する場合や、Web ページをジャンルに従ってラベル付けする場合、あるいは手書き文字を認識する場合などがあります。教師あり学習の学習者を作成するためには多くのアルゴリズムが使われ、最もよく使われるアルゴリズムとしてニューラル・ネットワーク、SVM (サポート・ベクトル・マシン)、単純ベイズ分類器などがあります。

教師なし学習では、ご想像のとおり、何が正しいか正しくないかの例を使わずにデータの意味を推測します。教師なし学習が最もよく使われる例としては、類似の入力を論理的なグループに分類する場合があります。また教師なし学習を使用してデータ・セットの次元数を削減することもできます。そうすると最も有用な属性のみに焦点を絞ることができ、あるいは傾向を検出することもできます。教師なし学習のための一般的な手法には、K 平均法、階層型クラスタリング、自己組織化マップなどがあります。

この記事では、現在 Mahout で実装されている下記の 3 つの機械学習タスクに焦点を絞ります。またこの 3 つは実際のアプリケーションでも非常によく使われるタスクです。

- 協調フィルタリング
- クラスタリング
- カテゴリー分け

これらの各タスクを概念的なレベルで詳しく説明した後、これらのタスクがどのように Mahout で実装されているかを説明します。

協調フィルタリング

協調フィルタリング (CF: Collaborative Filtering) は Amazon などで使われたことで知られるようになりました。CF ではユーザー情報 (評価、クリック数、購入履歴など) を使用して、他のサイト・

ユーザーに対してレコメンデーションを行います。CF は利用者に対して本や音楽、映画などを推薦するためによく使われますが、他のアプリケーションでも、複数のアクターが協力してデータを絞り込む場合などに使われます。おそらく皆さんも、Amazon で CF が実際に動作している様子を見たことがあるはずです (図 1)。

図 1. Amazon での協調フィルタリングの例

一連のユーザーとアイテムを指定すると、CF アプリケーションはシステムの現在のユーザーに対してレコメンデーションを行います。レコメンデーションを生成するための方法には通常、下記の 4 つがあります。

- **ユーザー・ベース**: 類似のユーザーを見つけることでアイテムを推薦します。ユーザーは動的なため、この方法は多くの場合、スケーラブルではありません。
- **アイテム・ベース**: アイテム間の類似性を計算し、レコメンデーションを行います。通常、アイテムが変更されることは少ないため、この計算はオフラインで行われることがよくあります。
- **スロープ・ワン**: 非常に高速で単純な、アイテム・ベースのレコメンデーション手法であり、ユーザーが評価を行ったときに適用可能です (単なるブール値のプリファレンス (好みの設定) ではありません)。
- **モデル・ベース**: ユーザー群とその評価のモデルを作成することでレコメンデーションを行います。

どの CF 手法も最終的には、ユーザーおよびユーザーが評価したアイテム同士の類似性の概念を計算することになります。類似性の計算方法は数多くあり、ほとんどの CF システムではさまざまな手法をプラグインできるようになっているため、どの手法が対象データに対して最も有効か、試しながら判断することができます。

クラスタリング

テキストであれ、数字であれ、大規模なデータ・セットがある場合、類似のアイテムを自動的にグループ化、つまりクラスタリングすると便利なことが多いものです。例えば、米国のすべての新聞による、ある日のすべてのニュースが与えられた場合、同じ話題に関するすべての記事を自動的にグループ化したいケースがあるかもしれません。そうすれば特定のクラスターとその話題に焦点を絞ることができ、大量の無関係な記事を処理する必要はなくなります。別の例として、

あるマシン上のセンサーからの長期間にわたる出力が与えられた場合、その出力をクラスタリングし、通常の操作と問題のある操作とを判別することができます。通常の操作の結果はすべてひとつのクラスターに集まり、問題のある操作は通常の操作とは別のクラスターに分散するからです。

CF の場合と同様、クラスタリングの場合も集合の中にあるアイテム同士の類似性を計算しますが、クラスタリングの仕事は類似のアイテムをグループ化することだけです。多くのクラスタリング実装では、集合の中のアイテムは n 次元空間のベクトルとして表現されます。ベクトル群が指定されると、マンハッタン距離、ユークリッド距離、あるいはコサイン類似度などの手法を使って 2 つのアイテムの間の距離を計算することができます。そして次に、距離が近いアイテム同士をグループ化することで実際のクラスターを計算します。

クラスターの計算手法は数多くあり、それぞれの手法には独特の長所と短所があります。一部の手法はボトムアップで動作し、小さなクラスターから大きなクラスターを構成していきますが、その一方、1 つの大きなクラスターを非常に小さなクラスターにまで分割する手法もあります。どちらの手法も、ある時点でそのプロセスを終了する条件があり、その後簡単なクラスター表現 (すべてのアイテムが 1 つのクラスターの中にある、あるいはすべてのアイテムがそれぞれ独自のクラスターにある) に分割されます。よく使われる手法には K 平均法と階層型クラスタリングがあります。後ほど説明するように、Mahout にはいくつかの異なるクラスタリング手法が用意されています。

カテゴリー分け

カテゴリー分け (分類と呼ばれることもよくあります) の目標は、未知の文書にラベルを付け、そうした文書をグループ分けすることです。機械学習での多くの分類手法では、さまざまな統計を計算することで文書の特徴を指定のラベルと関連付け、それによってモデルを作成し、後でそのモデルを使って未知の文書を分類できるようにします。例えば単純な分類手法として、あるラベルと関連付けられた単語群と、これらの単語群が指定のラベルに対して登場する回数を追跡します。そして新しい文書に対して分類が行われると、その文書の中の単語群がモデルの中で検索され、確率が計算されて、最善の結果が出力されます。通常はその結果の正しさの信頼度を示す数字も出力されます。

分類用の特徴に含まれるものには、単語群、単語群に対する (例えば頻度に基づく) 重み、品詞などがあります。もちろん、特徴は、ある文書をラベルと関連付けるものであれば、どのようなものでもよく、そうした特徴がアルゴリズムの中に取り入れられます。

機械学習の分野は幅広く、活発です。これ以上理論的な説明に焦点を絞るよりも (この記事ではとても説明しきれません)、Mahout とその使い方の説明に移ることにしましょう。

Mahout の紹介

Apache Mahout は ASF (Apache Software Foundation) による新しいオープンソース・プロジェクトであり、基本的な目標は Apache ライセンスの下、無料で使用できるスケーラブルな機械学習アルゴリズムを作成することです。このプロジェクトは 2 年目に入っており、既に 1 つのリリースが公開されています。Mahout には、クラスタリング、カテゴリー分け、CF、進化的プログラミングの実装が含まれています。しかも賢明なことに、Mahout は Apache Hadoop ライブラリーを使う

ことで、クラウドの中で Mahout を効果的にスケーリングできるようにしています (「参考文献」を参照)。

Mahout の歴史

Mahout の名前の由来

mahout というのは、象を飼う人や、象を使う人のことです。Mahout という名前の由来は、Mahout プロジェクトが (時によると) Apache Hadoop (ロゴは黄色い象) を使ってスケーラビリティーとフォルト・トレランスを実現していることによるものです。

Mahout プロジェクトは Apache Lucene (オープンソースの検索) コミュニティーに関係していた数人によって開始されました。彼らは機械学習に積極的な関心を持ち、クラスタリングとカテゴリー分けのための一般的な機械学習アルゴリズムを、確実かつ、十分にドキュメントを整備した上で、スケーラブルに実装しようとしていました。このコミュニティーは、Ng らによる論文「Map-Reduce for Machine Learning on Multicore (マルチコアでの機械学習のためのMap-Reduce)」(「参考文献」を参照) を契機として始まりましたが、その後、さらに幅広く機械学習手法をカバーするように進化してきました。Mahout にはその他にも、下記の目標があります。

- ユーザーやコントリビューターのコミュニティーを構築してサポートし、特定のコントリビューターがコミュニティーを去ったり、あるいは特定の企業や大学からの財政支援がなくなったりしてもコードの保守や改善を継続することができるようにする。
- 現実的で実用的な使い方に焦点を絞り、実績のない手法 (最先端の研究成果など) を避ける。
- 質の高いドキュメントとサンプルを提供する

機能

オープンソースの世界では比較的歴史が浅いものの、Mahout は既に (クラスタリングと CF に関しては特に) 大量の機能を持っています。Mahout の主な機能は以下のとおりです。

Map-Reduce について

Map-Reduce は Google で考案され、Apache Hadoop プロジェクトで実装された分散プログラミングのための API です。Map-Reduce を分散ファイルシステムと組み合わせると、プログラマーは、並列計算タスクを記述するために適切に定義された API を使用して、並列化問題を容易に扱えるようになることがよくあります (詳細は「参考文献」を参照)。

- Taste CF。Taste は CF のためのオープンソース・プロジェクトであり、SourceForge の Sean Owen によって開始され、2008年に Mahout に寄贈されました。
- Map-Reduce に対応した、いくつかのクラスタリング実装 (K 平均法、ファジィ K 平均法、Canopy、ディリクレ、平均シフトなど)。
- 分散型単純ベイズ分類器と補完型単純ベイズ分類器の実装。
- 進化的プログラミングのための分散型適応度関数機能。
- 行列ライブラリーとベクトル・ライブラリー。
- 上記のすべてのアルゴリズムのサンプル。

Mahout を使い始める

Mahout をセットアップして実行させる手順は比較的単純です。まず、必ず下記をインストールする必要があります。

- JDK 1.6 もしくはそれ以降のバージョン
- Ant 1.7 もしくはそれ以降のバージョン
- Mahout のソースをビルドしたい場合には Maven 2.0.9 または 2.0.10

また、この記事のサンプル・コードも必要です (「ダウンロード」を参照)。このサンプル・コードには、Mahout のコピーと依存関係ファイルが含まれています。以下の手順でサンプル・コードをインストールします。

1. `unzip sample.zip`
2. `cd apache-mahout-examples`
3. `ant install`

ステップ 3 によって、必要なウィキペディアのファイルがダウンロードされ、コードがコンパイルされます。ここで使用したウィキペディアのファイルは約 2.5 ギガバイトあるため、ダウンロード時間は帯域幅によって変わります。

レコメンデーション・エンジンを作成する

Mahout には現在、高速で柔軟な CF エンジンである Taste ライブラリーを使ってレコメンデーション・エンジンを作成するためのツールが用意されています。Taste はユーザー・ベースのレコメンデーションとアイテム・ベースのレコメンデーションの両方をサポートしており、またレコメンデーションを行うための多くの選択肢や、独自のレコメンデーションを定義するためのインターフェースも付属しています。Taste は下記の 5 つの基本コンポーネントから構成され、これらのコンポーネントが `User`、`Item`、`Preference` を処理します。

- `DataModel`: `User`、`Item`、`Preference` のためのストレージ
- `UserSimilarity`: 2 人のユーザーの間の類似性を定義するインターフェース
- `ItemSimilarity`: 2 つのアイテムの間の類似性を定義するインターフェース
- `Recommender`: レコメンデーションを提供するためのインターフェース
- `UserNeighborhood`: 類似のユーザーの近傍を計算するインターフェースであり、後で `Recommender` がこのインターフェースを使用します。

これらのコンポーネントとその実装によって、リアルタイム・ベースのレコメンデーションまたはオフラインのレコメンデーションのための複雑なレコメンデーション・システムを作成することができます。リアルタイム・ベースのレコメンデーションは数千人程度のユーザーしか処理できないことが多いのですが、オフラインのレコメンデーションは、それよりもはるかに多くのユーザーも処理することができます。Taste にはさらに、Hadoop を利用してオフラインでレコメンデーションを計算するためのツールまで用意されています。多くの場合、オフラインで計算する方法は、大量のユーザーやアイテム、プリファレンスがある大規模なシステムの要求を満たすための適切な方法です。

簡単なレコメンデーション・システムの作成方法を説明するには、何らかのユーザーとアイテム、評価が必要です。そのために、ここでは `cf.wikipedia.GenerateRatings` にあるコードを使用して、大量の `User` と `Preference` のセットをウィキペディアの文書 (Taste の用語での `Item`) に対してランダムに生成しました (このコードはサンプル・コードと共にソースの中に含まれています)。次にこのセットに対して、特定のトピック (Abraham Lincoln) に関して手動で作成した一連

の評価を追加し、サンプルの中に含まれている最終的な recommendations.txt を作成しました。このようにした理由は、特定のトピックのファンをそのトピックに関するその他の興味深い文書にガイドすることが CF によって可能になることを示すためです。サンプル・データの中には 990 人 (ラベルは 0 から 989) のランダムなユーザーが存在し、それらのユーザーは、集合の中のすべての記事に対してランダムに評価を付けています。また、別にユーザーが 10 人 (ラベルは 990 から 999) 存在し、この 10 人は、集合の中にある Abraham Lincoln という句を含む 17 の記事のうち 1 つ以上に評価を付けています。

でっち上げデータに注意

ここで示しているサンプル・データの中には完全にでっち上げのデータが含まれています。私は自分ですべての評価を行い、Abraham Lincoln に関する情報を好む実際のユーザー 10 人をシミュレートしました。このデータの背景にある概念は興味深いと思いますが、データそのものと、その値は興味深いものではありません。実際のデータが必要な場合には、University of Minnesota の GroupLens プロジェクトと Taste のドキュメントを調べてみることをお勧めします (「参考文献」を参照)。私がでっち上げのデータを選択した理由は、すべての例で 1 つのデータ・セットを使いたかったためです。

まず、recommendations.txt で一連の評価を行った 1 人のユーザーに対してレコメンデーションを作成する方法を説明します。Taste の使い方の典型的な例に従い、最初のステップとしてレコメンデーションを含むデータをロードし、そのデータを `DataModel` に保存します。Taste には、さまざまなファイルやデータベースを扱うための、いくつかの異なる `DataModel` の実装が含まれています。この例では簡単のため、FileDataModel クラスを使います。`FileDataModel` クラスは、各行の形式を、ユーザー ID、アイテム ID、プリファレンスと想定しています (ユーザー ID とアイテム ID は共にストリングですが、プリファレンスは倍精度の場合もあります)。モデルを用意できたら、どのようにユーザーを比較するのかを Taste に指示するために、`UserSimilarity` の実装を宣言します。使用する `UserSimilarity` 実装によっては、明示的な設定がされていないユーザーのプリファレンスをどのように推測するかも Taste に指示する必要があるかもしれません。以上の説明をコードに反映したものがリスト 1 です。(サンプル・コードの `cf.wikipedia.WikipediaTasteUserDemo` の中には完全なリストが含まれています。)

リスト 1. モデルを作成し、ユーザーの類似性を定義する

```
//create the data model
FileDataModel dataModel = new FileDataModel(new File(recsFile));
UserSimilarity userSimilarity = new PearsonCorrelationSimilarity(dataModel);
// Optional:
userSimilarity.setPreferenceInferrer(new AveragingPreferenceInferrer(dataModel));
```

リスト 1 では `PearsonCorrelationSimilarity` を使用して 2 つの変数の間の相関を測定していますが、他の `UserSimilarity` 尺度を利用することもできます。類似性の尺度に何を選択するかは、実際のデータとテストのタイプによって変わります。このデータの場合には、この組み合わせがベストであることが実際に相関の測定を行っている際にわかりました。どのように類似性の尺度を選択するかについては、Mahout の Web サイトを調べてください (「参考文献」を参照)。

このサンプルを完成させるために、`UserNeighborhood` と `Recommender` を作成します。`UserNeighborhood` は私のユーザーと似たユーザーを識別した後、`Recommender` に渡されます。すると Recommender は推薦されたアイテムを評価したリストを作成します。この考え方をコードに反映したものがリスト 2 です。

リスト 2. レコメンデーションを生成する

```
//Get a neighborhood of users
UserNeighborhood neighborhood =
        new NearestNUserNeighborhood(neighborhoodSize, userSimilarity, dataModel);
//Create the recommender
Recommender recommender =
        new GenericUserBasedRecommender(dataModel, neighborhood, userSimilarity);
User user = dataModel.getUser(userId);
System.out.println("-----");
System.out.println("User: " + user);
//Print out the users own preferences first
TasteUtils.printPreferences(user, handler.map);
//Get the top 5 recommendations
List<RecommendedItem> recommendations =
        recommender.recommend(userId, 5);
TasteUtils.printRecs(recommendations, handler.map);
```

このサンプル全体をコマンドラインで実行するためには、このサンプルを含むディレクトリーで `ant user-demo` を実行します。このコマンドを実行すると、Lincoln のファンである 995 という架空のユーザーに対するプリファレンスとレコメンデーションが出力されます。リスト 3 は `ant user-demo` を実行した場合の出力です。

リスト 3. ユーザーに対するレコメンデーションの出力

```
 [echo] Getting similar items for user: 995 with a neighborhood of 5
     [java] 09/08/20 08:13:51 INFO file.FileDataModel: Creating FileDataModel
            for file src/main/resources/recommendations.txt
     [java] 09/08/20 08:13:51 INFO file.FileDataModel: Reading file info...
     [java] 09/08/20 08:13:51 INFO file.FileDataModel: Processed 100000 lines
     [java] 09/08/20 08:13:51 INFO file.FileDataModel: Read lines: 111901
     [java] Data Model: Users: 1000 Items: 2284
     [java] -----
     [java] User: 995
     [java] Title: August 21 Rating: 3.930000066757202
     [java] Title: April Rating: 2.203000068664551
     [java] Title: April 11 Rating: 4.230000019073486
     [java] Title: Battle of Gettysburg Rating: 5.0
     [java] Title: Abraham Lincoln Rating: 4.739999771118164
     [java] Title: History of The Church of Jesus Christ of Latter-day Saints
              Rating: 3.430000066757202
     [java] Title: Boston Corbett Rating: 2.009999990463257
     [java] Title: Atlanta, Georgia Rating: 4.429999828338623
     [java] Recommendations:
     [java] Doc Id: 50575 Title: April 10 Score: 4.98
     [java] Doc Id: 134101348 Title: April 26 Score: 4.860541
     [java] Doc Id: 133445748 Title: Folklore of the United States Score: 4.4308662
     [java] Doc Id: 1193764 Title: Brigham Young Score: 4.404066
     [java] Doc Id: 2417937 Title: Andrew Johnson Score: 4.24178
```

リスト 3 の結果を見ると、このシステムが、さまざまなレベルの信頼度を持つ記事をいくつか推薦したことがわかります。実際、これらの各アイテムは他の Lincoln ファンによって評価されたものですが、995 というユーザーが評価したものではありません。他のユーザーの結果を見たい場合には、単純にコマンドラインで `-Duser.id=USER-ID` パラメーターを渡します (`USER-ID` は `0` から `999` の間の整数)。また `-Dneighbor.size=X` を渡すと近傍の大きさを変更することができます (`X` は 0 よりも大きな整数)。ちなみに、近傍の大きさを `10` に変更すると非常に異なる結果が得られますが、これはランダムなユーザーが 1 人近傍にいることによる影響です。複数のユーザーとアイテムに共通の近傍を表示するためには、`-Dcommon=true` をコマンドラインに渡します。

ここで、ユーザーの範囲外の数字を入力してしまった読者は、このサンプルが `NoSuchUserException` を出力することに気付いたかもしれません。実際には、アプリケーションは、新しいユーザーがシステムに参加してきた場合に処理すべき内容を実行する必要があったかもしれません。例えば、最も人気のある上位 10 位までの記事を単純に表示することもできますし、ランダムに選択した記事を表示したり、類似性のない記事を適当に選択して表示したり、あるいは何もしないようにすることもできます。

先ほど触れたように、ユーザー・ベースの手法は、あまりスケーラブルではありません。この場合にはアイテム同士の類似性を利用した手法を使った方が適切です。幸いなことに、この手法も Taste では簡単に使用することができ、そのセットアップと実行はユーザー・ベースの手法と大きく異なることはありません (リスト 4)。

リスト 4. アイテム同士の類似性を使う例 (`cf.wikipedia.WikipediaTasteItemItemDemo` より)

```
//create the data model
FileDataModel dataModel = new FileDataModel(new File(recsFile));
//Create an ItemSimilarity
ItemSimilarity itemSimilarity = new LogLikelihoodSimilarity(dataModel);
//Create an Item Based Recommender
ItemBasedRecommender recommender =
        new GenericItemBasedRecommender(dataModel, itemSimilarity);
//Get the recommendations
List<RecommendedItem> recommendations =
        recommender.recommend(userId, 5);
TasteUtils.printRecs(recommendations, handler.map);
```

リスト 1 の場合と同様、ここではレコメンデーション・ファイルから `DataModel` を作成していますが、この場合は `UserSimilarity` インスタンスをインスタンス化する代わりに `LogLikelihoodSimilarity` を使って `ItemSimilarity` を作成しています (`LogLikelihoodSimilarity` は稀にしか起こらない事象を処理するのに役立ちます)。そして `ItemSimilarity` を `ItemBasedRecommender` に渡し、レコメンデーションを要求しています。たったそれだけです。これをサンプル・コードの中で実行する場合は `ant item-demo` コマンドを使います。もちろん、ここから先の作業として、これらの計算をオフラインで実行するようにシステムをセットアップすることもでき、また他の `ItemSimilarity` 尺度を試すこともできます。ただし注意する点として、このサンプルではデータがランダムであるため、レコメンデーションは想定どおりにならないかもしれません。むしろ、テスト中に結果を評価し、異なる類似性メトリクスを試すということを確実に行うことが重要です。一般的な多くのメトリクスには、適切なレコメンデーションを行うほど十分なデータがない場合には破綻するような特定の例外ケースがあるからです。

では、新しいユーザーが参加してきた場合の例に戻りましょう。ユーザー・プリファレンスがない場合にどうするかという問題は、そのユーザーが 1 つのアイテムを選択してくれると、はるかに対処が容易になります。この場合には、アイテム同士の類似性を計算し、現在のアイテムに最も似ているアイテム群を `ItemBasedRecommender` に要求します。これをコードで示したものがリスト 5 です。

リスト 5. 類似のアイテムのデモ (`cf.wikipedia.WikipediaTasteItemRecDemo` より)

```
//create the data model
FileDataModel dataModel = new FileDataModel(new File(recsFile));
//Create an ItemSimilarity
ItemSimilarity itemSimilarity = new LogLikelihoodSimilarity(dataModel);
//Create an Item Based Recommender
ItemBasedRecommender recommender =
        new GenericItemBasedRecommender(dataModel, itemSimilarity);
//Get the recommendations for the Item
List<RecommendedItem> simItems
        = recommender.mostSimilarItems(itemId, numRecs);
TasteUtils.printRecs(simItems, handler.map);
```

リスト 5 をコマンドラインから実行するためには、`ant sim-item-demo` を実行します。リスト 4 とリスト 5 の本質的な唯一の違いは、レコメンデーションを要求するのではなく、入力アイテムに最も似ているアイテム群を要求している点です。

ここまで来ると、皆さん自身で Taste を深く掘り下げられるはずです。さらに詳しく学ぶためには、Taste のドキュメントと、mahout-user@lucene.apache.org のメーリング・リストを参照してください (「参考文献」を参照)。次に、Mahout のクラスタリング機能を活用して類似の記事を見つける方法を説明しましょう。

Mahout によるクラスタリング

Mahoutはいくつかのクラスタリング・アルゴリズムを実装しています。どのアルゴリズムも Map-Reduce を使って作成されており、それぞれが独自の目標と基準を持っています。

- Canopy: 高速のクラスタリング・アルゴリズムであり、他のクラスタリング・アルゴリズムのための出発点を作成するためによく使われます。
- K 平均法 (およびファジィ K 平均法): 前の繰り返しのセントロイド (つまり中心) からアイテム群までの距離を基に、アイテムを K 個のクラスターにクラスタリングします。
- 平均シフト: クラスターの数を事前に知る必要がなく、任意の形を形成するクラスターを生成できるアルゴリズムです。
- ディリクレ: 多くの確率論的モデルを組み合わせることでクラスタリングするため、クラスター群の特定のビューに初めから束縛されることがない、というメリットがあります。

現実的な視点で見ると、アルゴリズムの名前や実装は、そのアルゴリズムによって生成される結果ほど重要ではありません。それを念頭に置いた上で、ここでは K 平均法がどのように動作するかを説明し、他のアルゴリズムは皆さんが調べるように残しておきます。これらのアルゴリズムを効率的に実行させるためには、それぞれ独自の要件があることを忘れないでください。

大まかに言うと (詳細は後ほど説明します)、Mahout を使ってデータをクラスタリングするためのステップは以下のとおりです。

1. 入力を準備します。テキストをクラスタリングする場合には、そのテキストを数値表現に変換する必要があります。
2. Mahout に用意された Hadoop 対応の数多くのドライバー・プログラムの 1 つを使用して、適当なクラスタリング・アルゴリズムを実行します。

3. 結果を評価します。
4. 必要に応じて繰り返しを行います。

何よりも重要な点として、クラスタリング・アルゴリズムには、処理に適したフォーマットのデータが必要です。機械学習の場合、データは特徴ベクトルと呼ばれるベクトルとして表現されることがよくあります。クラスタリングでは、ベクトルというのはデータを表現する重みの配列です。ここではウィキペディアの文書から生成されたベクトルを使ってクラスタリングを説明しますが、センサー・データやユーザー・プロファイルなど他の領域からベクトルを得ることもできます。Mahout には、`DenseVector` と `SparseVector` という 2 つの `Vector` 表現が用意されています。適切なパフォーマンスを得るためには、データに応じて適切な実装を選択する必要があります。一般的に、テキスト・ベースの問題は疎 (sparse) であるため、そうした問題には `SparseVector` を選んだ方が適切です。その一方、大部分のベクトルのほとんどの値がゼロ以外の場合には `DenseVector` の方が適切です。どちらが適切かわからない場合には、データのサブセットに対して両方を試し、どちらの方が高速に処理が行われるかを調べる必要があります。

ウィキペディアのコンテンツからベクトルを生成するためには、以下を行います (ここでは既にベクトルを作成してあります)。

1. ベクトルの生成元としたいフィールドに対する単語ベクトル (term vector) を保存できるように、Lucene でコンテンツに索引付けをします。この詳細は、この記事で対象とする範囲を超えているため、説明を省略しますが、簡単なヒントと Lucene について少し説明しておきます。Lucene には (Lucene の `contrib/benchmark` パッケージの中に) `EnWikiDocMaker` というクラスがあり、このクラスがウィキペディア・ファイルのダンプを読み取り、Lucene での索引付け用の文書を生成することができます。
2. Lucene の索引から、Mahout の `utils` モジュールの中にある `org.apache.mahout.utils.vectors.lucene.Driver` クラスを使ってベクトルを作成します。このドライバー・プログラムには、ベクトルを作成するためのオプションが大量に用意されています。この詳細は Mahout のウィキ・ページ「Creating Vectors from Text (テキストからベクトルを作成する)」に説明されています (「参考文献」を参照)。

この 2 つのステップを実行すると、「Mahout を使い始める」セクションでダウンロードした n2.tar.gz ファイルのようなファイルが得られます。補足説明をすると、n2.tar.gz ファイルを構成しているのは、先ほどの `ant install` を実行することによって自動的にダウンロードされた、ウィキペディアの「チャンク」ファイルのすべての文書に索引付けをして得られたベクトルです。これらのベクトルはユークリッド・ノルム (または L^2 ノルム (「参考文献」を参照)) を使って正規化されています。Mahout を使う場合には、さまざまな方法でベクトルを作成し、どれによって最善の結果が得られるかを試してみたくなるはずです。

結果を評価する

クラスタリングした結果を評価するための手法はたくさんあります。多くの人は単純に、手動による検査と、特に計画性のないテストから始めます。しかし本当に満足できる結果を得るためには、通常はもっと詳細な評価手法を使う必要があります (例えば判断する人を数人使って明確な基準を作成するなど)。結果を評価する方法の詳細については「参考文献」を参照してください。この記事のサンプルでは、クラスタリングされた結果が実際に意味のあるものかどうかを手動による検査で調べています。これを本番に使用する場合には、これよりもはるかに厳密なプロセスを使う必要があります。

ベクトルのセットを作成できたら、次のステップとして K 平均法によるクラスタリング・アルゴリズムを実行します。Mahout には、`KMeansDriver` という適切な名前の付いた K 平均法アルゴリズムを始めとした、すべてのクラスタリング・アルゴリズムに対してドライバー・プログラムが用意されています。このドライバーは単純であり、例えば `ant k-means` を実行してみるとわかるように、Hadoop を使わないスタンドアロンのプログラムとして使用することができます。`KMeansDriver` がどのような引数を受け付けるかを調べるためには、build.xml の中で ant k-means ターゲットをいろいろ試してみてください。このプロセスが完了すると、その結果を `ant dump` コマンドを使って出力することができます。

スタンドアロン・モードで無事に実行できたら、Hadoop での分散モードでの実行に進むことができます。そのためには、サンプル・コードの hadoop ディレクトリーにある Mahout Job JAR が必要です。Job JAR にはすべてのコードと依存関係が 1 つの JAR ファイルにパッケージされ、容易に Hadoop にロードできるようになっています。また、Hadoop 0.20 もダウンロードして Hadoop のチュートリアルの指示に従う必要があります (このチュートリアルには、最初に疑似分散モード (つまりクラスターが 1 つ) で実行し、次に完全分散モードで実行する方法が説明されています)。詳細については、Hadoop の Web サイトやさまざまなリソース、そして IBM クラウド・コンピューティングのリソースを参照してください (「参考文献」を参照)。

Mahout を使ってコンテンツをカテゴリー分けする

Mahout は現在、ベイズ統計に基づいてコンテンツをカテゴリー分け (分類) する、2 つの関連する手法をサポートしています。第 1 の手法は Map-Reduce 対応の簡単な単純ベイズ分類器です。単純ベイズ分類器は、データが完全に独立しているという非常に単純な (そして多くの場合は不適切な) 前提で動作しますが、高速で非常に正確なことが知られています。単純ベイズ分類器は多くの場合、クラス当たりの訓練例の大きさのバランスが取れていないと、あるいはデータが十分に独立していないと、破綻します。補完型単純ベイズ分類器と呼ばれる第 2 の手法では、単純ベイズ分類器のいくつかの問題を修正しようとしていますが、単純ベイズ分類器の単純さと速さは維持されています。しかしこの記事では単純ベイズ分類器による手法のみを説明します。単純ベイズ分類器によって Mahout の全体的な問題と入力がわかるからです。

簡単に言えば、単純ベイズ分類器は 2 つの部分から成るプロセスであり、ある特定の文書とカテゴリーに関連付けられた特徴 (単語群) を追跡し、次にこの情報を使って新しい未知のコンテンツのカテゴリーを予測します。最初のステップは訓練と呼ばれ、モデルを作成します。そのために、既に分類されたコンテンツの例を調べ、各単語が特定のコンテンツに関連付けられている確率を追跡します。第 2 のステップは分類と呼ばれ、訓練中に作成されたモデルと、新しい未知の文書の内容、そしてベイズ理論を使用して、渡された文書のカテゴリーを予測します。従って、Mahout の分類器を実行するためには、まずモデルを訓練し、次のこのモデルを使って新しいコンテンツを分類する必要があります。次のセクションでは、これをウィキペディアのデータ・セットを使って行う方法を説明します。

単純ベイズ分類器を実行する

訓練器と分類器を実行する前に、少しばかり準備作業として、訓練用の文書セットとテスト用の文書セットをセットアップする必要があります。`ant prepare-docs` を実行し、(`install` ターゲットによってダウンロードしたファイルから) ウィキペディア・ファイルを準備します。そうすると、Mahout のサンプルに含まれている `WikipediaDatasetCreatorDriver` クラスを使ってウィキペディア入力ファイルが分割されます。文書群の分割は、関心対象のカテゴリー群の 1 つと一致

するカテゴリーが文書内にあるかどうかに基づいて行われます。関心対象のカテゴリーは、ウィキペディアの有効なカテゴリーであれば何でも構いません (さらには、ウィキペディアのカテゴリーのサブストリングであっても構いません)。例えば、この例には Science と History という 2 つのカテゴリーを含めてあります。従って、ウィキペディアのカテゴリーのうち、science または history という単語を含む (完全な一致である必要はありません) カテゴリーは、そのカテゴリーに関する他の文書群と共にバケットに入れられます。また各文書はトークン化され、正規化されることで、句読点や、ウィキペディアのマークアップ、そしてこのタスクに不必要なその他の特徴などが削除されます。最終的な結果は、カテゴリー名のラベルが付けられ、1 行ごとに 1 つの文書として 1 つのファイルに保存されます。これは Mahout が想定する入力フォーマットです。同様に、`ant prepare-test-docs` を実行すると、テスト文書に対して同じことが実行されます。テスト文書と訓練文書が重複しないことが重要です。重複すると適切な結果が得られません。理論的には、訓練文書をテストに使うと完璧な結果が得られるはずですが、それも現実的ではありません。

訓練用のセットとテスト用のセットが用意できたら、`ant train` ターゲットを使って `TrainClassifier` クラスを実行します。これにより、Mahout からも Hadoop からも大量のログが生成されるはずです。実行が完了すると、`ant test` は訓練中に作成されたモデルを使ってサンプルのテスト文書を分類しようとします。Mahout でそうしたテストを行うと、混同行列と呼ばれるデータ構造が出力されます。混同行列はカテゴリーそれぞれに対して、正しく分類された結果の数と、正しく分類されなかった結果の数を表現します。

要約すると、以下のステップを実行することによって分類結果が生成されます。

1. `ant prepare-docs`
2. `ant prepare-test-docs`
3. `ant train`
4. `ant test`

これらの 4 つをすべて実行すると (Ant ターゲット `classifier-example` を使うと、この 4 つを 1 回の呼び出しで実行することができます)、リスト 6 のような要約と混同行列が得られます。

リスト 6. history と science に対してベイズ分類器を実行した結果

```
[java] 09/07/22 18:10:45 INFO bayes.TestClassifier: history
                                95.458984375    3910/4096.0
[java] 09/07/22 18:10:46 INFO bayes.TestClassifier: science
                                15.554072096128172      233/1498.0
[java] 09/07/22 18:10:46 INFO bayes.TestClassifier: =================
[java] Summary
[java] -------------------------------------------------------
[java] Correctly Classified Instances          :       4143
                                                     74.0615%
[java] Incorrectly Classified Instances        :       1451
                                                     25.9385%
[java] Total Classified Instances              :       5594
[java]
[java] =======================================================
[java] Confusion Matrix
[java] -------------------------------------------------------
[java] a           b       <--Classified as
[java] 3910        186      |  4096        a     = history
[java] 1265        233      |  1498        b     = science
[java] Default Category: unknown: 2
```

中間プロセスの結果は、基本ディレクトリー内の wikipedia というディレクトリーに保存されます。

結果セットが得られると、「どうすればよかったのか」という当然の質問が涌きます。この Summary を見ると、約 75 パーセントが正しく分類され、25 パーセントは正しく分類されなかったことがわかります。一見したところでは、これは非常に妥当な数字に思えます。ランダムに推測した場合よりも結果が良いため、特にそう感じられます。しかし詳しく調べてみると、history に関する予測は非常に良く (約 95 パーセントを正しく分類)、science に関する予測は非常にお粗末です (約 15 パーセント)。この理由を調べるために、訓練用の入力ファイルを見てみると、history のサンプルは science のサンプルよりも大幅に多いことがわかります (ファイル・サイズはほぼ倍です)。ここに 1 つの問題がありそうです。

テストを行う場合、`ant test` に `-Dverbose=true` を追加すると、各テスト入力に関する情報と、そのテスト入力のラベル付けが適当であったかどうかが出力されます。この出力を調べてみると、その文書がなぜ正しく分類されないのか、文書を調べて手掛かりを検証することができます。また、異なる入力パラメーターを試したり、science のデータを増やしてモデルを維持し、結果を改善できるかどうかを調べたりすることもできます。

また、モデルを訓練するための特徴の選択についても考えることが重要です。ここで説明したサンプルでは Apache Lucene の `WikipediaTokenizer` を使って元の文書をトークン化しましたが、不適切にトークン化される可能性のある共通の単語や価値のない単語を削減する努力をあまりしていません。この分類器を本番に使用する場合には、入力や他の設定を十分深く検証し、可能な限り高いパフォーマンスが得られるように努力する必要があります。

science の結果が偶然だったかどうかを調べるために、ここでは Republicans (共和党) と Democrats (民主党) という別のカテゴリー・セットを試してみました。この場合には、新しい文書が Republicans に関するものか Democrats に関するものかを予測します。これを皆さん自身で試せるように、src/test/resources の中に repubs-dems.txt を作成しました。次に、以下のような分類ステップを実行します。

```
ant classifier-example -Dcategories.file=./src/test/resources/repubs-dems.txt -Dcat.dir=rd
```

`-D` が付いた 2 つの値は、単にカテゴリー・ファイルと、wikipedia ディレクトリー配下に中間結果を入れるためのディレクトリーの名前を指定しています。これを実行した結果の要約と混同行列はリスト 7 のようになります。

リスト 7. Republicans と Democrats に対してベイズ分類器を実行した結果

```
 [java] 09/07/23 17:06:38 INFO bayes.TestClassifier: --------------
 [java] 09/07/23 17:06:38 INFO bayes.TestClassifier: Testing:
                              wikipedia/rd/prepared-test/democrats.txt
 [java] 09/07/23 17:06:38 INFO bayes.TestClassifier: democrats      70.0
                                                                    21/30.0
 [java] 09/07/23 17:06:38 INFO bayes.TestClassifier: --------------
 [java] 09/07/23 17:06:38 INFO bayes.TestClassifier: Testing:
                             wikipedia/rd/prepared-test/republicans.txt
 [java] 09/07/23 17:06:38 INFO bayes.TestClassifier: republicans    81.3953488372093
                                                                    35/43.0
 [java] 09/07/23 17:06:38 INFO bayes.TestClassifier:
```

```
[java] Summary
[java] -------------------------------------------------------
[java] Correctly Classified Instances          :         56           76.7123%
[java] Incorrectly Classified Instances        :         17           23.2877%
[java] Total Classified Instances              :         73
[java]
[java] =======================================================
[java] Confusion Matrix
[java] -------------------------------------------------------
[java] a           b       <--Classified as
[java] 21          9        |  30          a     = democrats
[java] 8           35       |  43          b     = republicans
[java] Default Category: unknown: 2
```

最終的な結果は、正しさという点ではほとんど同じですが、こちらの方が 2 つのカテゴリーを適切に判断できていることがわかります。入力文書を含む wikipedia/rd/prepared ディレクトリーを簡単に調べてみると、この 2 つの訓練用ファイルの方が訓練のためのサンプルに関してバランスがとれていることがわかります。またこのディレクトリーを調べた結果から、history/science を実行した場合と比べて全体として大幅にサンプルが少ないこともわかります。これは各ファイルが history または science の訓練用セットよりもはるかに小さいためです。全体として、少なくとも結果は history/science の場合よりも大幅にバランスが取れています。訓練用セットを大きくすると、おそらく Republicans と Democrats のバランスが取れて差が少なくなるはずですが、そうならないとしても、それはいずれか一方がその党のメッセージをうまくウィキペディアに追加しているためかもしれません。しかしその件については政治学者の判断に任せることにします。

これで分類器をスタンドアロン・モードで実行する方法を説明できました。次のステップはコードをクラウドに適用し、Hadoop クラスターで分類器を実行することです。それには、クラスタリング・コードの場合と同様、Mahout Job JAR が必要です。それさえ用意すれば、先ほど触れたすべてのアルゴリズムは Map-Reduce に対応しているため、Hadoop のチュートリアルの中で概説されている Job 送信プロセスの下で実行する場合には問題なく動作するはずです。

Mahout での次の一歩

Apache Mahout は、ほんの 1 年と少しの間に大きく前進し、クラスタリング、カテゴリー分け、CF の機能が大幅に高まりましたが、まだ成長の余地は多分にあります。直近の課題としては、分類を決定するためのランダム・フォレストを Map-Reduce で実装することや、関連付けルール、文書内のトピック識別のための潜在的ディリクレ配分法 (Latent Dirichlet Allocation)、HBase やその他のバッキング・ストレージ・オプションを使ってカテゴリー分けする方法の検討などがあります。これらを新たに実装することの他に、デモやドキュメントを増やすことや、バグを修正することなどもあります。

最後に、本物の象使いが象の強さと能力を活用するのと同様に、Apache Mahout を使うことで、黄色い象、つまり Apache Hadoop の強力さと能力を活用することができます。次回、皆さんがコンテンツのクラスタリングやカテゴリー分け、あるいはレコメンデーションをする場合には、特にそれを大規模に行う場合には、Apache Mahout を検討してみてください。

謝辞

この記事を校閲してくださり、助言をいただいた、Mahout のコミッターである同僚、Ted Dunning と Sean Owen の両氏に感謝いたします。

ダウンロード

内容	ファイル名	サイズ
Sample code	j-mahout.zip	90MB

著者について

Grant Ingersoll

Grant Ingersoll は Lucid Imagination の設立者であり、技術スタッフの 1 人でもあります。彼は情報取得、機械学習、テキストのカテゴリー分け、抽出などなどのプログラミングに関心を持っています。彼は Apache Mahout 機械学習プロジェクトの共同設立者の 1 人であり、Apache Lucene プロジェクトと Apache Solr プロジェクト両方のコミッターでもあり、講演者でもあります。また彼は自然言語処理のためのオープンソース・ツールを解説する『Taming Text』(Manning より出版予定) の共著者の 1 人でもあります。