事前に考慮していない要因が大きく影響をあたえる場合も有ります。そこでオンラインでの評価として、モデルを適用しない場合のパフォーマンスや、複数のモデルによる予測結果を使った場合のパフォーマンスを比較することで、より最適なモデルを選択することができます。

また、A/Bテストできるようなシステム構成にしておくことで、副次的に新しいモデルの段階的なリリースや切り戻しも可能となります。そうすることで、検証のイテレーションサイクルを素早く回すことが可能となりますし、機会損失を抑えることができます。

オフラインで複数のアルゴリズムやパラメータチューニングを行ったモデルを用意し、A/Bテストで選別をし、更に良いモデルを作成しオフラインで検証し、A/Bテストに投入する、という検証のサイクルを回していくのが良いでしょう。詳しくは、6章を参照してください。

3.4 この章のまとめ

本章では学習結果の評価方法について学びました。

分類の指標として、正解率や適合率、再現率、F値について学びました。実際には混同行列を見ながら、どのクラスがどれくらいの性能であれば良いかを考えていくのが重要です。

回帰の評価指標として、平均二乗誤差と決定係数について学びました。どちらの指標を用いても良いですが、何を基準とするかという点について常に気をつけましょう。

また、機械学習におけるA/Bテストの重要性も学びました。特に、機械学習の評価指標の良し悪しと、ビジネス上のゴールとしてのKPIの良し悪しとは別になってきます。この2つの違いを常に意識することで、予測モデルの評価指標のみを追いかけてしまわないように気をつけましょう。オフラインで評価指標の目標を達成することは、機械学習を使ったビジネスのスタートラインに立つための最低条件です。

4章 システムに機械学習を組み込む

機械学習をシステムに組み込むにはどうすればいいでしょうか。本章では、機械学習を組み込むシステムの構成とそれに深く関わる教師データを獲得するためのログ収集方法について説明します。

4.1 システムに機械学習を含める流れ

「1.2 機械学習プロジェクトの流れ」でも書きましたが、システムに実際に機械学習を適用する際には、以下のような流れで進めていきます。

1. 問題を定式化する
2. 機械学習をしないで良い方法を考える
3. システム設計・誤りをカバーする方法を考える
4. アルゴリズムを選定する
5. 特徴量、教師データとログの設計をする
6. 前処理をする
7. 学習・パラメータチューニング
8. システムに組み込む

本章では、この中でも「システム設計」「ログ設計」について説明をしていきます。

4.2 システム設計

機械学習にはいくつかの種類がありますが、ここでは最も活用ケースが多い教師あり学習について、システムに組み込む場合の構成を説明します。

分類や回帰などの教師あり学習の場合、学習と予測の2つのフェーズがあります。更に学習のタイミングによって、バッチ処理での学習とリアルタイム処理での学習という二種類のタイミングがあります。

本節で、それぞれの場合のシステム構成とそのポイントについて学びますが、まずその前に重要でありながら混乱しがちな用語について整理しておきましょう。

4.2.1 混乱しやすい「バッチ処理」と「バッチ学習」

機械学習において、「バッチ」という言葉は特別な意味を持ちます。いわゆるバッチ処理と語源は同じですが、多くの場合、機械学習の文脈で「バッチ」というと「バッチ学習」のことを指します。ここでは「バッチ学習」と一般的な「バッチ処理」との違いについて説明します。

本章では、バッチ処理の対義語をリアルタイム処理と呼ぶことにします。バッチ処理は一括で何かを処理すること、またその処理そのものを指します。対してリアルタイム処理は、刻々と流れてくるセンサーデータやログデータに対して逐次処理をすること、と本書では定義します。[†1]

なお、本章では「バッチ処理」との混同を避けるために、これ以降はバッチ学習を**一括学習**、オンライン学習を**逐次学習**と表現します。

一括学習と逐次学習とでは、モデル学習時のデータの保持の仕方が異なります。一括学習では、重みの計算のためにすべての教師データを必要とし、全データを用いて最適な重みを計算します。一般的に一括学習の場合、教師データが増えると必要とするメモリ量はその分増加していきます。例えば、求める重みw_targetがすべての重みの平均だったとします。重みがw_1からw_100まで100個ある時の平均を一括学習で求めるには以下のように計算します。

```
sum = w_1 + w_2 + w_3 + ... + w_100
w_target = sum / 100
```

一方、逐次学習では教師データを1つ与えて、その都度重みを計算します。例えば、次のような平均の求める処理をする場合、メモリに保持されるデータはその時のデータと、計算された重みのみになります。ある時点での重みがw_tmpとしたとき、平均を計算するのに必要なのは総和sumと、要素の数cntだけです。コードで表現すると以下のようになります。

```
sum = 0
cnt = 0
while has_weight():
  w_tmp = get_weight()
  sum += w_temp
  cnt += 1

w_target = sum / cnt
```

繰り返しになりますが、一括学習と逐次学習では学習時の必要とするデータの塊が違います。つまり、学習時の最適化の方針が違うだけなのです。[†2]

では、バッチ処理は何を処理するのでしょうか？実は「バッチ処理」というだけでは、特に規定されていません。機械学習の文脈では学習をすることもあるし、予測をすることもあります。リアルタイム処理も同様に、学習をすることもありますし予測をすることもあります。

ここで問題です。以下の組み合わせの中で、取りうる処理と学習の組み合わせはどれでしょうか？

1. バッチ処理で一括学習
2. バッチ処理で逐次学習
3. リアルタイム処理で一括学習
4. リアルタイム処理で逐次学習

良くある誤解は「一括学習はバッチ処理でしかできず、逐次学習はリアルタイム処理でしかできない」というものです。実は3以外はすべてありえます。1と4について、特に違和感はないかもしれません。では、2の「バッチ処理で逐次学習」とはどのようなものでしょうか？逐次学習は、最適化時にデータを1レコードずつ処理をする最適化方針だと説明しました。つまりバッチ処理でまとまったデータを一括処理をするけれど、最適化方針は逐次学習するということはあり得るのです。

†1 「リアルタイム処理」というと、「何msで処理ができるの？」と思われる方もいるかもしれませんが、本書では便宜上速度に関係なく逐次的な処理をすることをリアルタイム処理と呼んでいます。

†2 なお、一括学習と逐次学習の中間となるミニバッチ学習(mini-batch training)という方法もあります。ある程度のデータをサンプリングしたグループを作り、このグループに対する一括学習を繰り返します。確率的勾配法(SGD)が有効だと知られてから急速にミニバッチ学習が広まりました。深層学習ではミニバッチ学習が使われることが主流です。

予測フェーズについては、学習フェーズでの最適化の方針や処理方法にかかわらずバッチ処理での予測もリアルタイム処理での予測も共に存在します。

実際に学習をする際は、データを保持できない場合を除いて学習フェーズはバッチ処理でするのが試行錯誤しやすくて良いでしょう。

ここからは、バッチ処理で学習を行う3つの予測パターンとリアルタイム処理のパターンについて構成を見て行きましょう。以下にパターンを列挙します。

1. バッチ処理で学習＋予測結果をWebアプリケーションで直接算出する（リアルタイム処理で予測）
2. バッチ処理で学習＋予測結果をAPI経由で利用する（リアルタイム処理で予測）
3. バッチ処理で学習＋予測結果をDB経由で利用する（バッチ処理で予測）
4. リアルタイム処理で学習をする

4.2.2 バッチ処理で学習＋予測結果をWebアプリケーションで直接算出する（リアルタイム処理で予測）

3つの予測パターンの中で、最も素朴な方法がこのパターンです。このパターンは、バッチ処理で一括学習をし、そこで得られた予測モデルをWebアプリケーションでリアルタイム処理で利用するというものです。monolithic[†3]なWebアプリケーションに予測処理を組み込み、予測結果はライブラリのAPIから取得し、それをWebアプリケーションに渡します(図4-1)。

†3 バッチシステムとWebアプリケーションなど複数の機能や役割をまとめた一つの大きなシステムをとるアーキテクチャ

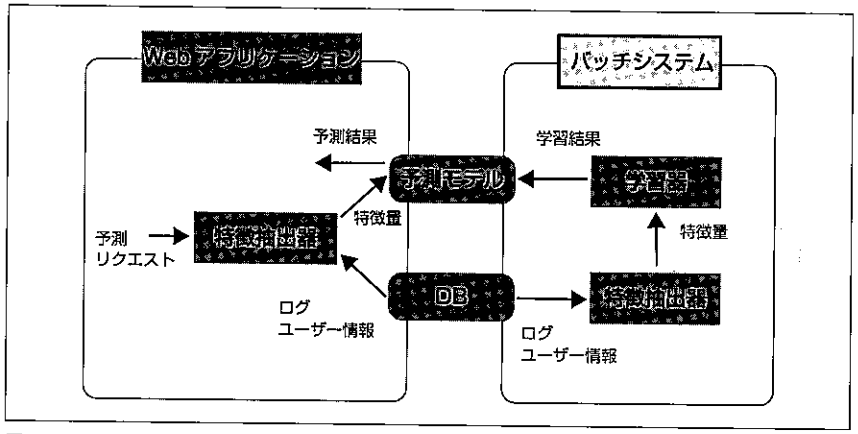

図4-1 パターン1: バッチ処理で学習したモデルを直接Webアプリケーションで使う

このパターンの特徴としては

- 予測はリアルタイム処理が必要
- Webアプリケーションと機械学習をするバッチシステムの言語が同一

という2点があります。

このパターンは比較的単純な構成のため試すのも容易で、小規模で試してみるのに適したパターンです。

バンディットアルゴリズムを用いた広告配信の最適化など、入力データが事前に用意できず、予測の結果を低レイテンシに使いたい場合にもこの構成が取られます。レイテンシを抑えるためには、データのフェッチ、前処理、特徴抽出、予測といった一連の処理が低レイテンシで完結することが望ましいのです。そのためRDBやKey Value Storeなどのデータベースに、予め前処理済みのデータや、特徴抽出の工程を減らすためにある程度の段階までの処理を終えた特徴量を格納しておくなどの工夫を施します。また、予測モデルもメモリに容易に載せられるサイズで、予測処理の負荷が低いアルゴリズムというように、空間計算量、時間計算量の観点でコンパクトなモデルが望ましいでしょう。

一方で、機械学習の処理部分とWebアプリケーションが密結合になりやすいという側面があります。アプリケーションが大規模になると、コード変更やデプロイのコストが増えて、機械学習部分の開発も保守的になりがちです。

この制約を嫌って、機械学習のプロトタイプはPythonで行い、Webアプリケーションで利用しているJavaScriptやRubyへ予測ロジックを移植したり、C++で書かれたライブラリのバインディングを作成したりする場合もあります。

このシステム（図4-1）では、Webアプリケーションとバッチシステムが同じ言語で組まれています。ほとんどの部分をWebアプリケーションとバッチシステムで共有して使いまわします。1つのDBから取得したログやユーザー情報から（図では別のモジュールになっていますが）共通の特徴抽出器を用いて特徴量を抽出します。「1.2.5 特徴量、教師データとログの設計をする」で学んだ通り、特徴抽出器は、テキストなどの情報から学習器が理解できる形に変換をする部分でした。この特徴抽出の処理が異なると、いかに同じ学習済みモデルを用いても同じ予測結果にはなりません。

ログ設計については「4.3 ログ設計」で詳しく説明します。

学習フェーズ（図4-2）では、バッチシステムによってDBから予め蓄積されたログやユーザー情報を取得し、特徴量を抽出します。ここで得た特徴量をもとに何らかのモデルを学習します。学習結果は学習済みモデルをシリアライズして保存したものをストレージに保持します。

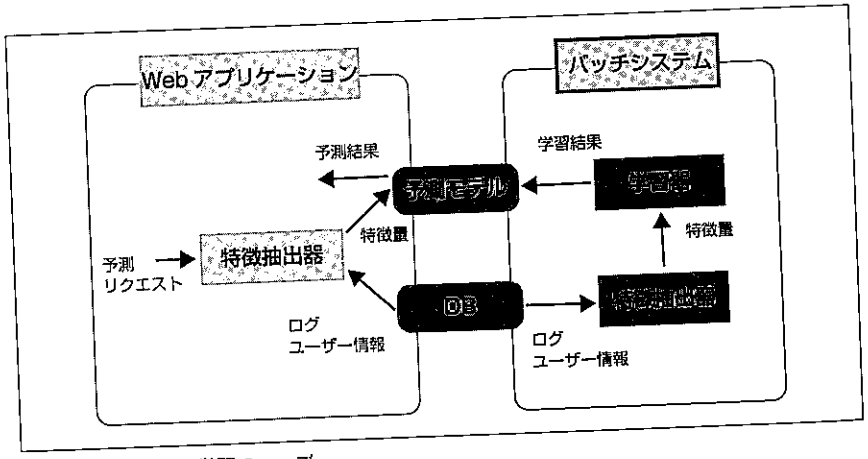

図4-2 パターン1: 学習フェーズ

予測フェーズ（図4-3）になると、Webアプリケーションが何かしらのイベントをトリガーに予測を要求します。例えば、スパムかどうかを判定したいコメントが投稿されたとします。イベント発生時に予測をしたい対象（例：コメント）の情報をDBから（あるいはリクエスト情報から直接）取得し、特徴量を抽出します。シリアライズして保存していた学習済みモデルを読み込み、抽出した特徴量を入力して予測結果（例：スパム/非スパム）を出力します。その結果を元に、ユーザーにフィードバックをするなどして、次の処理へとつなげていきます。

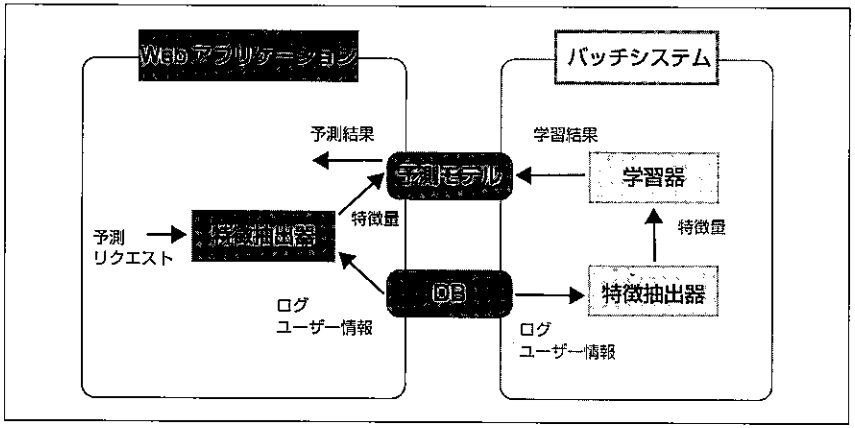

図4-3 パターン1: 予測フェーズ

4.2.3 バッチ処理で学習＋予測結果をAPI経由で利用する（リアルタイム処理で予測）

Webアプリケーションとは別に、予測処理を薄くラップしたAPIサーバーを用意するのがこのパターンです（図4-4）。このパターンでは、バッチ処理で学習を行うことは他のパターンとは代わりませんが、Webアプリケーションから予測結果を利用する場合にはAPI経由のリアルタイム処理で予測を行います。HTTPやRPCのリクエストに対して、予測結果をレスポンスとして返すAPIサーバーを用意するのが特徴です。

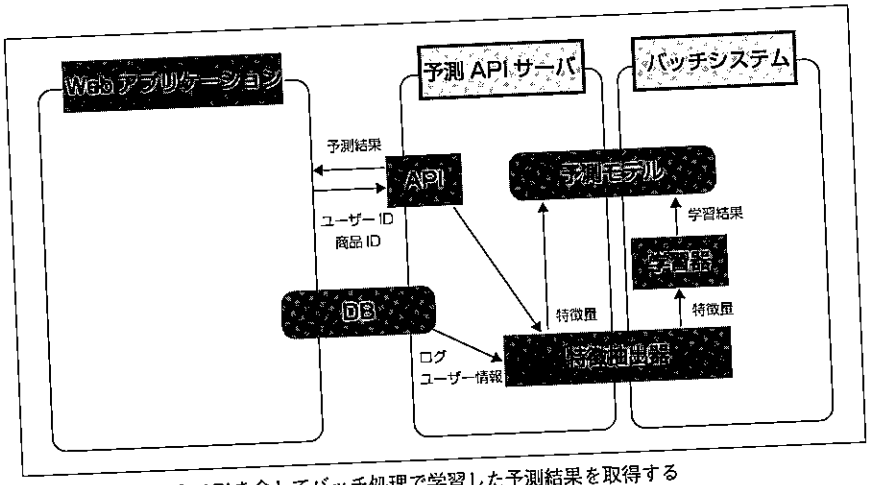

図4-4 パターン2: APIを介してバッチ処理で学習した予測結果を取得する

このパターンの特徴は、

- Webアプリケーションと機械学習に使うプログラミング言語を分けられる
- Webアプリケーション側のイベントに対してリアルタイム処理で予測できる

というものです。

機械学習環境を自由に選べるためプロトタイピングを高速に回せる反面システムの規模が大きくなるため、リアルタイム処理での予測が重要でない場合には取りづらい構成です。scikit-learnなどのライブラリを使って構築するには自前でAPIサーバーを実装し、予測サーバの前にロードバランサを配置し、負荷に応じて予測サーバを増減できるようにするといった、スケールするような工夫が必要です。もし、手軽に試してみたい場合は、Azure Machine Learning、Amazon Machine Learningなどの機械学習サービスや、Apache PredictionIO (incubating)などの予測サーバーまで含んだフレームワークを利用するという方法もあります。最近ではAWS Lambdaを使った予測APIを作ることで、イベント駆動でスケールしやすい予測を行うことも容易になってきました。また、予測モデルをAmazon S3などのオブジェクトストレージに格納し、APIサーバのDocker image作成することで、Amazon Elastic Container ServiceやGoogle Kubernetes Engineを使い、スケールしやすい構成も組みやすくなっています。

このパターンを採用すると、Webアプリケーションとの結合が疎になることから、学習に使うアルゴリズムや特徴量を変えた複数のモデルによるA/Bテストを行う場合に、

モデル間の比較がしやすいといったメリットもあります。

ただし、パターン1に比べるとAPIサーバと予測結果を利用するクライアントの間で通信が発生する分、レイテンシが大きくなることに注意してください。そのため、レイテンシをより小さくしたい場合は、HTTPやRPCなどのリクエストを投げる部分を非同期処理にして、予測処理の結果を待つ間に他の処理を並列で進めるなどの工夫を行うと良いでしょう。

4.2.4 バッチ処理で学習+予測結果をDB経由で利用する(バッチ処理で予測)

Webアプリケーションで使い勝手の良いのはこのパターンです。一番はじめに試すパターンとしてはこの方法が無難でしょう。

分類問題などについて教師あり学習のモデルを一括学習し、そのモデルを使った予測をバッチ処理で行い、その予測結果をDBに格納するという方法です。

このパターンは、予測バッチとアプリケーションの間でDBを介してやりとりをするため、Webアプリケーションと機械学習の学習・予測を行う言語がそれぞれ異なっていても良いことが大きなメリットです。また、後述するAPIパターンとは異なり、予測の処理に多少時間がかかる場合でもアプリケーションのレスポンスに影響しません。

このパターンの特徴は、以下の通りです。

- 予測に必要な情報は予測バッチ実行時に存在する
- イベント(例:ユーザのWebページ訪問)をトリガーとして即時に予測結果を返す必要がない

具体的には、商品説明など変化のしにくいコンテンツを6時間毎のバッチで分類する、ある日のユーザー閲覧履歴からどのユーザークラスターに所属するかを日次バッチで処理する、といったように、予測の頻度がおよそ一日一回以上(短くても数時間に一回)程度で問題のない対象や結果に向いています。例えば、ユーザのアクセスログからメールマガジンで送付する内容をパーソナライズする、などがこれに該当します。

このパターンのシステム構成は図4-5のようになります。Webアプリケーションと機械学習を行うバッチシステムとのやりとりはDBのみを介して行うため、両者のシステムに言語的な依存関係は特に発生しません。つまりWebアプリケーションでRuby on Railsを使っていたとしても、特に気にすることなくPythonやRでバッチを書けるた

め、アルゴリズムの選定や特徴選択など機械学習の試行錯誤のサイクルがより高速に回せます。

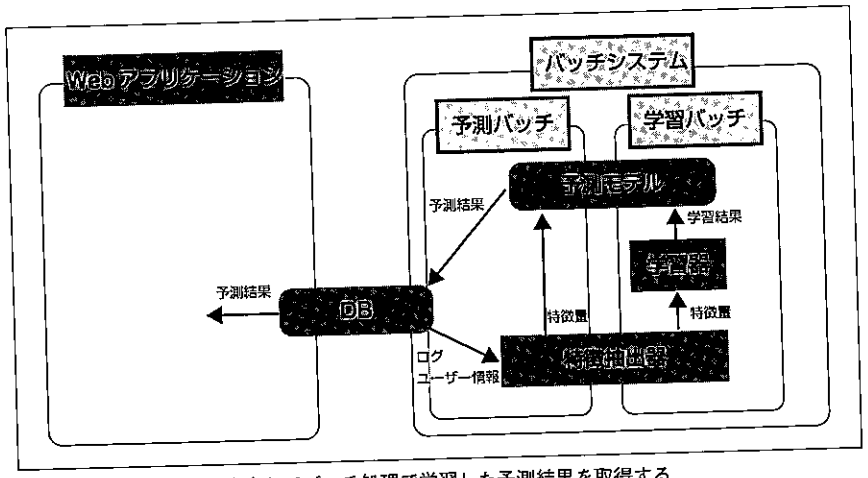

図4-5 パターン3: DBを介してバッチ処理で学習した予測結果を取得する

学習フェーズ（図4-6）では、ログやユーザー情報から特徴を抽出してモデルを一括学習します。ここで構築した学習済みモデルは、シリアライズしてストレージに保持し予測フェーズで使用します。

学習バッチの実行間隔は予測の間隔よりも広くとります。再学習を行う間隔は、予測対象がどの程度変化するかに依存します。定期的に再学習する場合は、「1.2.8 システムに組み込む」で紹介した、ゴールドスタンダードを利用するなど、学習しなおした後に精度が低下していないことを確認する工夫が必要です。

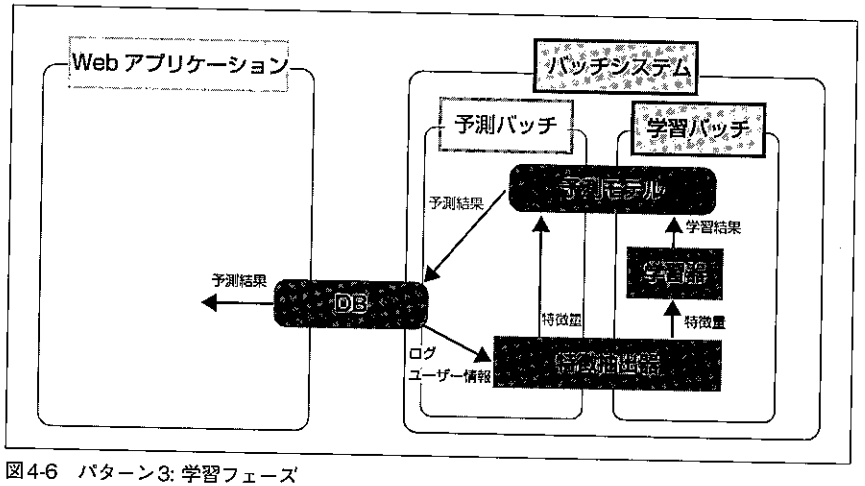

図4-6 パターン3: 学習フェーズ

予測フェーズ（図4-7）では、学習バッチで作成したモデルを用いて予測を行います。学習バッチと同様の特徴抽出器を用いて、DB中のデータから特徴量を抽出し、予測します。予測結果はWebアプリケーションが利用できる形にしてDBへ格納します。

このパターンは他のパターンに比べ、予測にかけられる時間に余裕があるのが特徴ですが、予測対象となるコンテンツが増えていくと、それに比例して処理時間も増えていきます。そのため、全コンテンツに対して予測し直すようなバッチの組み方をすると、データ量の増加に対して処理時間が予想以上に膨らんでしまい、日次のジョブでは終わらないようなことが起きるので注意が必要です。

特に、モデルを頻繁に再学習したり、使用する特徴量やアルゴリズムを変えて複数モデルを作成したりする場合は予測にかかる時間に十分注意する必要があります。もし、データの特性がそこまで大きく変化しないことが保証されているであれば、新規で登録された差分のコンテンツに対してのみ予測を行うという戦略を取ることもできます。すべてのデータに対して予測し直す必要がある場合は、並列数を増やして予測処理を行うか、Sparkなどの分散処理が可能な環境で予測するのが良いでしょう。

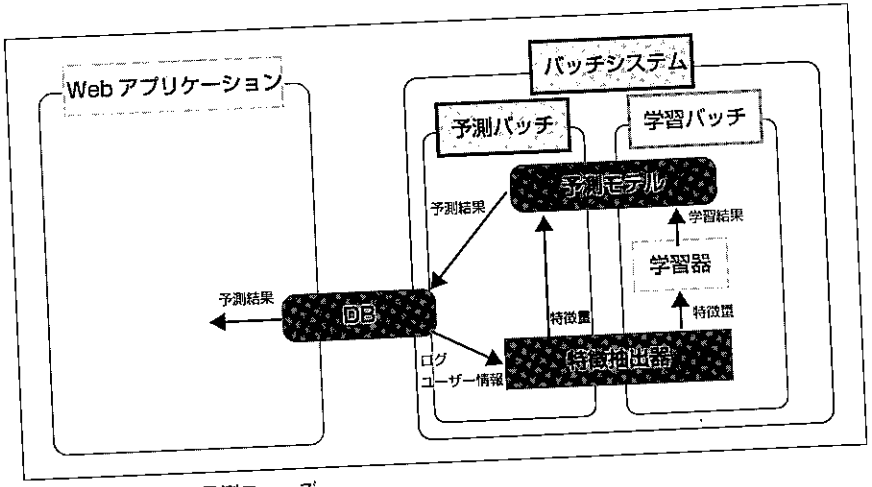

図4-7 パターン3: 予測フェーズ

4.2.5 リアルタイム処理で学習をする

「4.2.1 混乱しやすい「バッチ処理」と「バッチ学習」」では、リアルタイム処理での学習はないと言いましたが、実は全くないわけではありません。リアルタイム処理で学習が必要な場合とは、どのような場合でしょうか。

バンディットアルゴリズムなど一部のアルゴリズムやリアルタイムレコメンドでは、リアルタイム処理を使って即時にパラメータの更新が必要となる場合があります。その場合、メッセージキューなどを使い入出力のデータをやりとりします。しかし分類や回帰などで、そこまで即時にモデルを更新する必要がある場合は多くないでしょう。

もし、ある程度短い間隔でモデルを更新する必要がある場合は、1時間おきなど任意のタイミングで蓄えたデータに対してバッチ処理で学習をし、最適化方針は追加学習のできるミニバッチ学習を採用するという方法が良いでしょう。

リアルタイムレコメンドの構成としては、Oryx[†4]という、リアルタイムの更新を分散メッセージキューであるApache Kafkaと組み合わせたフレームワークがあります。こちらのアーキテクチャが参考になるでしょう。また、Jubatus[†5]という逐次学習向けのフレームワークも、このパターンで使われることを想定したものになります。

†4 https://github.com/OryxProject/oryx
†5 http://jubat.us/ja/

4.2.6 各パターンのまとめ

各パターンの特徴を表4-1にまとめます。

表4-1 システム構成のパターンのまとめ

パターン	一括学習+直接予測	一括学習+API	一括学習+DB	リアルタイム
予測	リクエスト時	リクエスト時	バッチ	リクエスト時
予測結果の提供	プロセス内API経由	REST API経由	共有DB経由	MQ経由
予測リクエストから結果までのレイテンシ	○	○	◎	◎
新規データ取得から予測結果を渡すまでの時間	短	短	長	短
一件の予測処理にかけられる時間	短	短	長	短
Webアプリケーションとの結合度	密	疎	疎	疎
Webアプリケーションのプログラミング言語と	同一	独立	独立	独立

選択の際には、特にWebアプリケーションと独立した機械学習のライブラリが充実した言語での開発と、データ取得から予測結果を返すまでのサイクルの時間のトレードオフが重要になってきます。

開発のスピードと処理速度のトレードオフを考え、適切なパターンを選んで下さい。

Pythonで学習したモデルを Python以外で利用する

豊富なアルゴリズムやユーザーの多さから、いまやデファクトスタンダードとなっているscikit-learnですが、scikit-learnで学習をしたモデルを別の言語で扱えるようにするという事例もあるようです。筆者が知る範囲では、SwiftやJavaScriptでモデルを扱えるようにしたという事例があります。

後者については実装した人に直接聞いたところ、Webアプリケーション側のコードがNode.jsだったため、決定木やロジスティック回帰といったアルゴリズムをNode.jsで再実装しているそうです。[†6]機械学習のライブラリを実装するの

†6 http://www.slideshare.net/TokorotenNakayama/mlct

は普通のプログラミングよりバグの検証が困難で、なかなか厳しい道だと思います。

こうした問題を解決するため、PMML[†7]やPFA[†8]といった、言語やフレームワークをまたいでモデルをインポート／エクスポートするための規格もありますが、サポートしているフレームワークも限定的で2017年の段階で銀の弾丸とはなり得ていません。

またTensorFlowはPythonのAPIも備えたフレームワークですが、TensorFlow Liteの形式にモデルを変換することでiOSやAndroidでも学習済みのモデルを使うことができます[†9]。今後、フレームワークレベルで複数のプラットフォームをサポートするソフトウェアが増えてくるかもしれません。またAppleは、iOS 11からCore MLと呼ばれるiOS向けのフレームワークを用意しています。特筆すべきは、scikit-learnやXGBoost、Kerasなど様々な機械学習フレームワークで学習したモデルをiOS向けに変換することができるようになったことです。[†10]これにより学習したモデルをiOS用に変換し、高速に予測できるようになることが期待されます。

4.3 ログ設計

本節では、機械学習システムの教師データを取得するためのログの設計と、特徴量について説明します。

機械学習、特に教師あり学習を行う場合、Webサーバのアプリケーションログや、どこをどうクリックしたかなどのユーザの行動ログなどを集めて、そこから特徴量を抽出します。

機械学習の入力となる教師データはシステムのログから作成するのが一般的です。

†7 http://dmg.org/pmml/v4-3/GeneralStructure.html
†8 http://dmg.org/pfa/index.html
†9 https://www.tensorflow.org/mobile/
†10 https://developer.apple.com/documentation/coreml/converting_trained_models_to_core_ml

ログは、DBなどのデータと異なりスキーマがない、記録してないデータを後から改めて取得するのが困難、といった特徴があり、システムに組み込むにあたっては様々なコツがあります。ログ設計は特徴量を決めるための重要なポイントです。例えば、複数の自社で展開するWebサービスごとにユーザIDが異なる場合、CookieなどにUUIDを仕込んでIDの名寄せを試みるなどをする必要があります。しかし、UUIDを記録していなければユーザIDもマッピングできないため、複数のサービスをまたいだ特徴量を得られません。特徴量は、Feature Engineeringという言葉があるように試行錯誤が必要なものですが、ログにない情報を作る工夫をするよりはログをあらかじめ仕込めるのであれば、そちらの方が簡単です。考えうる必要なログを取得するためにも、どういった情報が必要かを考えましょう。

本節では、どこにあるどういった情報を利用し、教師データに活用するかについての概要を説明します。なお、具体的な教師データの詳細な収集方法は5章にて説明します。

4.3.1 特徴量や教師データに使いうる情報

特徴量や教師データに使えそうな情報としては、大きく以下の3つがあります。

1. ユーザー情報
2. コンテンツ情報
3. ユーザー行動ログ

ユーザー情報は、ユーザーに登録してもらう時に設定してもらう、例えば性別のようなユーザーの属性情報のことです。**コンテンツ情報**は、ブログサービスにおけるブログ記事や商品などのコンテンツ自身の情報です。これらは一般的には、MySQLを中心としたOLTP (Online Transaction Processing) 向けのRDBMSに保持されています。**ユーザー行動ログ**は、ユーザーがどのページにアクセスしたか（アクセスログ）やユーザーが商品の購入などイベントを起こしたことのログです。特にユーザー行動ログは、広告のクリックイベントや商品の購買などコンバージョンに繋がる情報を持つことが多く、教師データになりやすいので、適切に収集できるようにしましょう。ユーザー行動ログはデータ量が多くなるので、オブジェクトストレージや分散RDBMS、Hadoop上のストレージなどに保存することが多いです。

4.3.2 ログを保持する場所

ユーザー行動ログはデータ量が多くなるので、保存場所には気をつける必要があります。MySQLやPostgreSQLなど業務用のRDBMSに格納すると、後々全データの傾向を見たりして当たりをつけることが難しくなります。こうしたデータは、機械学習用途だけでなく、レポーティングやダッシュボードなど、集計処理を経て可視化されることも多くあります。機械学習の前に対話的な分析をすることを踏まえると、以下のようなデータの保持方法が考えられます。

- 分散RDBMSに格納する
- 分散処理基盤Hadoopクラスターの HDFSに格納する
- オブジェクトストレージに格納する

これらの保持方法に共通しておすすめなのは、SQLでデータにアクセスできるようにすることです。SQLでデータにアクセスできるようにしておくと、他のプログラミング言語を書かなくても様々な分析ができます。データの中の必要な情報を選別した上で転送をするといった操作が容易になるので、データの転送コストが下がります。近年では、Amazonの Amazon RedshiftやGoogleの Google BigQueryなどのフルマネージドなクラウド型の分散DBサービスが展開されており、いわゆるデータウェアハウスを手軽に用意できるようになりました。

あるいはApache Hadoopを用いた分散ファイルシステムHDFS(Hadoop Distributed File System) に格納するのも良いでしょう。Apach Hive、Apache Impala(Incubating)、PrestoなどHadoop上で動くSQLクエリエンジンを用いることで、SQLを使ったデータアクセスが容易になります。

2つ目と似ていますが、クラウドストレージに直接格納するのも選択肢の1つです。その場合は、Amazon Elastic MapReduce (EMR) やGoogle Cloud Dataproc、Azure HDInsightなどマネージドな分散処理サービスを使うことで、SQLやMapReduceだけでなくApache Sparkを使った複雑な処理も可能です。特に近年では、Amazon S3のようなオブジェクトストレージにデータを格納して、そこに対してImpalaやHive、PrestoやAWS Athenaでクエリを直接実行するといったスタイルも増えてきました。

これらの生のデータからSQLを使った集計処理等をした後、機械学習用のデータセットとして利用します。

既にWebアプリケーションを運用している場合、クラウドストレージや分散データベースにアクセスログなどのログデータを保管していると思います。下記のようなマネージドのクラウドサービスを利用することで、管理コストが低減されるでしょう。

- クラウドストレージ
  - Amazon S3
  - Google Cloud Storage
  - Microsoft Azure BLOB Storage
- マネージド分散DB
  - Amazon Redshift
  - Google BigQuery
  - Treasure Data

こうしたログは、Fluentdや Apache Flume、Logstashといったログ収集ソフトウェアをWebアプリケーションサーバーに入れて、保管先に転送します。また、最近ではEmbulkのようなバッチでデータを転送するソフトウェアや、分散メッセージングシステムApache Kafkaを活用してスケーラブルなログ収集基盤を作る、というように選択肢が増えています。

4.3.3 ログを設計する上での注意点

機械学習を含んだシステムの開発を進めるときに、特徴量を抽出するには試行錯誤を繰り返すことがほとんどで、最初から有効な特徴量を見つけるのは困難です。つまり、必要そうなユーザー情報、コンテンツ情報についてはサービス設計時にあらかじめ想定しておく必要があります。

KPIの設計時にはできるだけ少ない指標にする方が良いのですが、機械学習に使える情報は出来る限り多い方が望ましいのです。後から必要に応じて特徴選択のロジックを加えたり次元圧縮をしたりすることは可能ですが、保存していない情報を増やすのは困難です。例えばあるユーザーが広告をクリックするかを予測する際に、性別や午前中／夕方に訪問した、掲出する広告のカテゴリなど、予測に関する情報の多様性を確保する必要があります。

また、現在取得しているログで教師データを作れるか、という視点も必要です。実際にあった例としては「広告をクリックしたログ」は保存していたが「広告を表示したログ」はデータ量が多すぎるため破棄していた、ということがありました。この場合「広

告が表示されたがクリックされなかった」というログが存在しないことになるため、教師データがうまく作れず、クリック予測が行えませんでした。

このほかにも、マスターデータの変更履歴を保存していなかった、という事例もありました。商品の説明文と売れ行きの関係を調査してほしいという依頼があり、購買ログと商品マスターを受け取りました。しかし、商品の説明文の変更は、商品マスターを直接書き換えて運用しており、いつからいつまでどのような説明文で販売していたのか、という情報が欠落していました。そのため、十分な調査を行うことができませんでした。

このように、システム開発・運用をする人と分析をする人が分かれていると、検証に必要なコンバージョンしなかったなどのネガティブなデータや、マスターデータの変更履歴など重要なデータを捨ててしまう事があるので注意が必要です。

もう1つ気をつけて欲しいのが、ログ形式の変化についてです。機械学習をするにはデータ量が多いほうが満足の行く性能に達する可能性が上がります。長い期間のデータを集める上で、サービスの機能追加や仕様変更により、ログの形式が変化し取得する情報が変わる場合があります。しかし、入力に使う特徴量のセットを途中で変化させることはまずありません。従って、長い期間の古い情報量の少ない時の特徴量のセットを使うか、短い期間で新しい特徴量のセットを使うかどちらが良いか検討が必要です。

大規模データの転送コスト

大規模データの機械学習における最も大きなボトルネックは、データの転送時間です。筆者の経験では、1GBを超えたログの生データを一括でダウンロードしてオンメモリで処理をするのは、やめたほうが良いと思います。

scikit-learnを使った学習をする場合に、どうしても機械学習のバッチ処理を行うサーバーにデータを転送しなければならないのですが、その時間を抑えるためには、分散RDBMSを利用したデータウェアハウスにMySQLなどのOLTPサーバーのデータを同期し、出来る限り分散RDBMS上でSQLを使って前処理をできるようにするのが望ましいでしょう。

大規模なデータに対して、複雑な前処理を定期的に実行する必要がある場合は、例えばAmazon S3に置いたデータをAmazon Elastic Map Reduceで加工するなど、出来る限りローカルマシンにダウンロードしない工夫をすることが必要です。

4.4 この章のまとめ

本章では、機械学習を情報システムに組み込むための設計と、ログ設計について説明しました。

一括学習をして得られたモデルから、予測結果をどのように呼び出すかによって4つのパターンがあります。

- バッチ処理で学習＋予測結果をWebアプリケーションで直接算出する（リアルタイム処理で予測）
- バッチ処理で学習＋予測結果をDB経由で利用する（バッチ処理で予測）
- バッチ処理で学習＋予測結果をAPI経由で利用する（リアルタイム処理で予測）
- リアルタイム処理で学習

ログ設計に合わせて特徴量や教師データをすることが重要になってきます。これらを考えるときは、できるだけ手戻りが少なくなるように設計しましょう。