マシンラーニング向け線形モデルの高速化

この記事は、Medium の Intel Analytics Software で公開されている「Accelerating Linear Models for Machine Learning」の日本語参考訳です。

線形回帰がこれまでになく高速に

Python* と scikit-learn を使用して大規模なデータセットからマシンラーニング (ML) モデルを構築した経験がある場合、これらの計算を高速化したいと思われたことがあるかもしれません。1 行のコードを変更するだけで ML の計算を高速化できるとしたらどうしますか? また、より高速に結果を得るため特別なハードウェアを必要としないとしたらどうでしょう?

この記事では、インテル® CPU 向けに最適化された scikit-learn (英語) を使用してリッジ回帰モデルをトレーニングする方法を紹介し、標準の scikit-learn ライブラリーでトレーニングされたモデルのパフォーマンスおよび精度と比較します。この記事は、高速な ML アルゴリズム・シリーズの 1 つです。

線形回帰の実践例

特殊なリッジ回帰である線形回帰にはさまざまな用途があります。ここでは、良く知られている Kaggle (英語) の House Sales in King County, USA (米国キング郡の住宅販売) (英語) データセットを使用します。このデータセットを使用して、キング郡の 1 年間の住宅販売データを基に住宅価格を予測してみます。

このデータセットには、21,613 行と 21 列があります。各行は、2014 年 5 月から 2015 年 5 月までにキング郡で販売された住宅を表します。第 1 列には販売を識別する一意の ID、第 2 列には住宅の販売日、そして第 3 列には販売価格 (ターゲット変数) が含まれています。第 4 列から第 21 列には、寝室数、広さ、築年数、郵便番号など、家に関するさまざまな数値的特徴が含まれています。これらの情報を基に、住宅の販売価格を予測するリッジ回帰モデルを構築します。

理論的には、線形回帰モデルの係数は最小残差平方和 (RSS) にすべきです。実際には、最小 RSS のモデルが必ずしも最良であるとは限りません。線形回帰は、入力データが多重共線性を持つ場合、不正確なモデルを生成することがあります。この場合、リッジ回帰はより信頼性の高い予測を提供できます。

scikit-learn で回帰問題を解く

sklearn.linear_model.Ridge でモデルを構築する方法を説明します。次のプログラムは、住宅販売データセットの 80% の行でトレーニングを行い、残り 20% の行でモデルの精度をテストします。

```
import numpy as np
import pandas as pd
from sklearn import config_context
from sklearn.linear_model import Ridge
from sklearn.model_selection import train_test_split

# Load the data from a text file into pandas DataFrames.
# The third column contains the responses. Load it into df_y.
# Columns after the fourth contain features. Load them into df_X.

infile = "data/kc_house_data.csv"
df_X = pd.read_csv(infile, usecols=range(3, 20))
df_y = pd.read_csv(infile, usecols=[2])

# Split the data into training and testing subsets.
# Use 80% of the rows to train the regression model.
# Use the remaining 20% to test the model.

X_train_np, X_test_np, y_train_np, y_test_np =    \
    train_test_split(df_X, df_y, train_size=0.8, \
                     random_state=7777777)

# Convert the data into pandas DataFrames for convenience:

X_train = pd.DataFrame(X_train_np)
y_train = pd.DataFrame(y_train_np)
X_test = pd.DataFrame(X_test_np)
y_test = pd.DataFrame(y_test_np)

# Construct a ridge regression algorithm object:
alg = Ridge(alpha=1.0)

# Train a ridge regression model:

model = alg.fit(X_train, y_train)

# Check the model's accuracy on a test set:

y_pred = model.predict(X_test)

MSE = ((y_test.values - y_pred)**2).sum()
RMSD = np.sqrt(MSE/y_test.size)
R2 = alg.score(X_test, y_test)
```

結果は 0.69 になりました。つまり、構築したモデルはデータ分散の 69% を表しています。トレーニング済みモデルの品質に関する詳細は、「**付録**」を参照してください。

インテルにより最適化された scikit-learn

リッジ回帰によるトレーニングと予測時間は非常に高速ですが、ハイパーパラメーターをチューニングするため、通常、トレーニングを複数回実行する必要があります。また、予測精度を向上させるため、さまざまな特徴のサブセットでモデルを評価してみるとよいでしょう。大規模なデータセットでは 1 回のトレーニングに数分かかることもあり、トレーニングを複数回実行するには長時間を要するため、線形モデルのトレーニングのパフォーマンスは重要です。

Python* 向けインテル® ディストリビューションは、ネイティブ Python* のドロップイン置換であり、インテル® アーキテクチャー向けに最適化されています。conda や pip などの一般的なチャネルからインストール可能なパッケージとシームレスに動作します。Python* 向けインテル® ディストリビューションの scikit-learn には、Continuum の scikit-learn と同じアルゴリズムと API のセットが含まれているため、ML アルゴリズムのパフォーマンスを向上するためコードを変更する必要はありません。

また、daal4py (英語) (インテル® oneAPI データ・アナリティクス・ライブラリー (インテル® oneDAL) (英語) の Python* インターフェイス) を利用することで、scikit-learn のパフォーマンスをさらに向上することが可能です。daal4py は、構成可能な ML カーネルを提供し、その一部はストリーミング入力データをサポートしており、ワークステーションのクラスターに容易にスケーリングできます。

scikit-learn と daal4py の設定方法

Python* 向けインテル® ディストリビューションをインストールする方法はいくつかあります。conda でインストールするには、こちらの手順に従ってください。

次のコマンドは、'idp' conda 環境に daal4py をインストールします。

```
conda install -n idp daal4py
```

ソルバーとして daal4py を使用するには、scikit-learn の動的なパッチを適用する必要がありますが、アプリケーションを変更することなくパッチを有効にできます。次のコマンドライン・オプションを使用します。

```
python -m daal4py my_app.py
```

アプリケーションでパッチを有効にすることもできます。

```
import daal4py.sklearn
daal4py.sklearn.patch_sklearn()
```

パッチを元に戻すには、次のコマンドを実行します。

```
daal4py.sklearn.unpatch_sklearn()
```

パッチを適用すると、次の scikit-learn アルゴリズムに影響します。

- sklearn.linear_model.LinearRegression (英語)
- sklearn.linear_model.Ridge (英語) (solver='auto')
- sklearn.linear_model.LogisticRegression (英語) と sklearn.linear_model.LogisticRegressionCV (英語) (['lbfgs', 'newton-cg'] のソルバー)
- sklearn.decomposition.PCA (英語) (svd_solver='full'、svd_solver='daal' の追加)
- sklearn.cluster.KMeans (英語) (algo='full')
- sklearn.metric.pairwise_distance (英語) (metric='cosine' または metric='correlation' の場合)
- sklearn.svm.SVC (英語)

このリストは、Python* 向けインテル® ディストリビューションの次のリリースで拡張される予定です。

パフォーマンスの比較

標準の scikit-learn とインテルにより最適化された scikit-learn のパフォーマンスを比較するため、キング郡のデータセットとサンプル数と特徴量が異なる 6 つの人工的に生成されたデータセットを使用しました。人工的なデータセットは、scikit-learn make_regression 関数を使用して生成しました。

```
X, y = make_regression(n_samples=nrows, n_features=ncols, \
                       n_informative=ncols, noise=10.0, bias=10.0, \
                       random_state=7777777)
X_train = pd.DataFrame(X)
y_train = pd.DataFrame(y)
```

図 1 は、次の 2 つの構成でリッジ回帰モデルのトレーニングにかかったウォールクロック時間です。

- デフォルトの conda チャネルからインストールした scikit-learn 0.22
- daal4py で最適化された Python* 向けインテル® ディストリビューションの scikit-learn 0.21.3

インテル® oneDAL の scikit-learn 最適化を有効にするため、-m daal4py コマンドライン・オプションを使用しました。パフォーマンスの測定には、Amazon Web Services* Elastic Compute Cloud* (AWS* EC2*) を使用しました。最高のパフォーマンスを提供する次のインスタンスを選択しました。

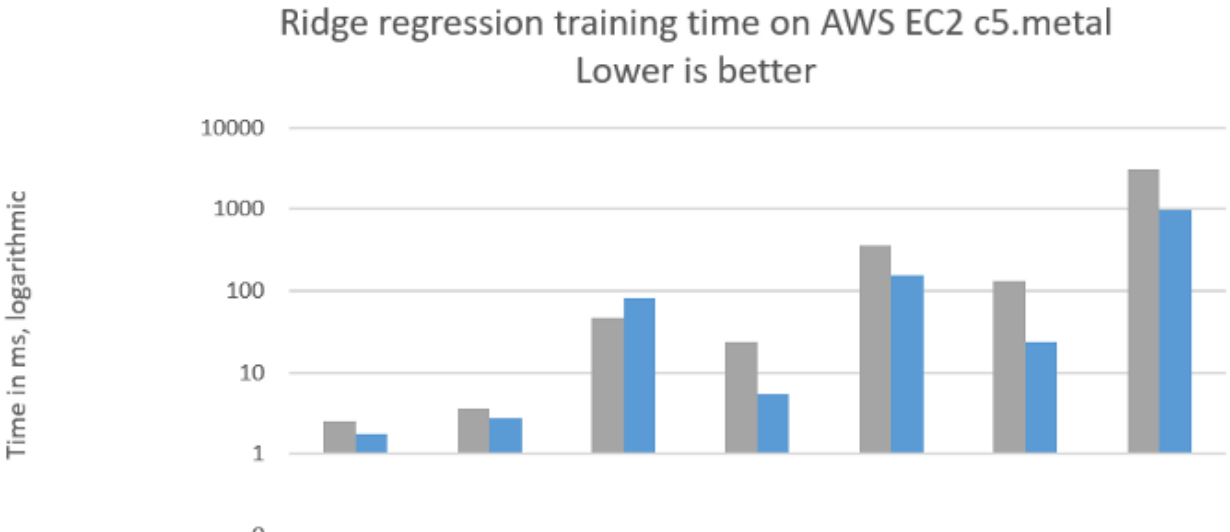
- **CPU:** c5.metal (第 2 世代インテル® Xeon® スケーラブル・プロセッサー、2 ソケット、ソケットごとに 24 コア)。

以下は、Amazon* のウェブサイトからの抜粋です。

> Amazon* EC2* C5 インスタンスは、高度なコンピューティング集約型ワークロードの実行のための費用対効果の高いハイパフォーマンスを、低額のコンピューティング比率あたりの料金で提供します。

c5.metal インスタンスは、C5 インスタンスの中で最も多くの CPU コア数と最も新しい CPU を搭載しています。

ハードウェアの詳細は、「**システム構成**」セクションを参照してください。トレーニング済みモデルの品質に関する詳細は、「**付録**」を参照してください。



Data set Number of rows Number of features	House 17290 17	artificial 10000 100	artificial 10000 1000	artificial 100000 100	artificial 100000 1000	artificial 1000000 100	artificial 1000000 1000
Vanilla sklearn	2.54	3.57	47.12	23.53	359.45	132.04	3144.44
Intel-optimized sklearn	1.77	2.78	82.37	5.53	153.17	24.07	983.65
Vanilla Sklearn / Intel-optimized Sklearn	1.44	1.28	0.57	4.26	2.35	5.49	3.20

クリックして拡大

図 1. リッジ回帰のトレーニング時間

図 1 から、次のことが分かります。

- インテルにより最適化された scikit-learn は、標準の scikit-learn と比較して、**リッジ回帰のトレーニングが最大 5.49 倍高速**です。
- データセットのサイズに応じて、**インテルにより最適化された scikit-learn のパフォーマンスの向上は大きくなります**。

Python* 向けインテル® ディストリビューションの scikit-learn が高速な理由

大規模なデータセットでは、リッジ回帰はほとんどの計算時間を行列乗算に費やします。インテル® oneDAL のリッジ回帰実装は、インテル® CPU 向けに高度に最適化されたインテル® マス・カーネル・ライブラリー (インテル® MKL) を利用しています。インテル® MKL は、第 2 世代インテル® Xeon® スケーラブル・プロセッサーで利用可能なインテル® アドバンスト・ベクトル・エクステンション 512 (インテル® AVX-512) の SIMD (Single Instruction Multiple Data) ベクトル命令を使用します。行列乗算などの計算集約型カーネルは、これらの命令によるデータ並列性から大きな恩恵を受けます。さらに、行列をブロック分割してインテル® スレッディング・ビルディング・ブロック (インテル® TBB) で並列に処理することで、行列乗算において別のレベルの並列性が得られます。

インテルにより最適化された scikit-learn は、モデルの精度を損なわず、わずかなコード変更でリッジ回帰のパフォーマンスを大幅に向上します。パフォーマンスの優位性は、このアルゴリズムに限ったことではありません。前述のように、最適化された ML アルゴリズムのリストは拡張し続けています。

システム構成

ハードウェア

c5.metal AWS EC2 instance: Intel Xeon 8275CL processor, two sockets with 24 cores per socket, 192 GB RAM. OS: Ubuntu 18.04.3 LTS.

Testing date: 04/06/2020

ソフトウェア

Vanilla sklearn: Python 3.8.0, scikit-learn 0.22, Pandas 0.25.3.

Intel Distribution for Python scikit-learn: Conda install from intel channel: Python 3.7.4, scikit-learn 0.21.3 optimized with daal4py 2020.0 build py37ha68da19_8, Pandas 0.25.1.

Python and accompanying libraries are the default versions installed by the conda package manager when configuring the respective environments.

付録

Ridge Regression Model Accuracy						
Data Set	Number of rows	Number of columns	RMSD		R^2	
			Vanilla scikit-learn	Intel-optimized scikit-learn	Vanilla scikit-learn	Intel-optimized scikit-learn
King County	17,290	17	200,221.014	200,221.014	0.69414	0.69414
make_regression	10,000	100	10.018	10.018	0.99969	0.99969
make_regression	10,000	1,000	9.609	9.609	0.99997	0.99997
make_regression	100,000	100	9.996	9.996	0.99970	0.99970
make_regression	100,000	1,000	9.987	9.987	0.99997	0.99997
make_regression	1,000,000	100	10.014	10.014	0.99969	0.99969
make_regression	1,000,000	1,000	9.995	9.995	0.99997	0.99997

クリックして拡大

表 1 リッジ回帰モデルの平均二乗偏差 (RMSD) と決定係数 (R^2)