

Special Contents

データサイエンティスト 中級者への道

この章では、データサイエンティスト入門レベルを卒業し、中級者の道に進むためのさまざまな分析の手法やアプローチ、ツールについて紹介します。

一部実装もありますが、最低限の基礎的な説明や概念の紹介が中心となります。深層学習を学ぶために必要となる基礎知識、Pythonの処理を高速化するためのツール群、膨大なデータをサーバーに分散処理させるためのSparkの紹介など、今後データ分析をする上で身につけておきたいスキルばかりです。

はじめは理解しにくい箇所もあるかもしれませんが、このようなアプローチやツールがあるということを知っておくだけでも、今後の学習にきっと役に立つはずです。なお、本章は基本的に紹介のみですので、用語の解説やサンプルコードは少ししかありません。ここで紹介した手法やツール等については、あくまでリファレンス程度のもので、これらのことを本格的に学びたい場合は、本章だけのコンテンツでは不十分です。参考文献等を読んだり、他の講座で勉強して、どんどんスキルを磨いていってください。

Goal データサイエンティスト中級者になるためのさまざまなアプローチやツールを知る（深層学習の基礎知識の習得、Pythonの処理で並立処理やCython、Spark等を活用して高速化するための方法を知る）

Special 1

この章の概要

Keyword 深層学習、ディープラーニング、パーセプトロン、勾配降下法、誤差逆伝播法、前処理、Cython、Spark

この章ではまず、最近注目されている深層学習（ディープラーニング）を学ぶための前準備として、いくつか押さえておきたい基礎概念や実装を紹介します。この講座で学んだ統計知識（最尤法、最小二乗法など）や機械学習の考え方（教師あり学習、教師なし学習、交差検証法、混同行列など）と、ここで学ぶ概念（パーセプトロン、勾配降下法、誤差逆伝播法など）を勉強しておけば、深層学習を学ぶ前の良い準備となるでしょう。

次に、機械学習や深層学習等のモデリングの話から離れて、Pythonの処理を高速化するための方法をいくつか紹介します。

モデル作成の前に、データ加工をする必要（いわゆる前処理）が多々ありますが、その前処理の計算時間を改善することも大事です。もちろん、コーディングスキルによって、計算処理時間は異なってくるのですが、そのアルゴリズム作成にも限界があります。ここで紹介するライブラリ等を使うことで、計算コストを下げることができます。

一昔前、Pythonはスクリプト言語だから遅いという指摘もありました。ただ、昨今、Cythonや並列処理などの色々なライブラリが出てきており、下手なCプログラムを書くよりも、Pythonで実装したほうがいいこともあります。すべての高速化処理について紹介することはできず、ここで紹介する実装が必ず最適というわけではありませんが、計算時間に課題があったときには、ぜひここで使うライブラリ等を使って計算速度をあげることも検討してください。

さらに、分散処理をするためのSparkについて説明します。

ご承知の通り、現代の世の中には大量のデータがあり、そのデータは日々蓄積されています。この講座でもある程度、こうしたビッグデータに対応できるようなスキルを身に付けるため、いろいろなツールやアプローチを紹介してきました。ここではさらに、そうした膨大なデータに対して、データの加工から機械学習まで一貫して処理し、複数のサーバーを使った分散処理で計算スピードを上げ、さらにリアルタイムに分析ができるSparkを紹介します。今回は入門ということで、その機能と使い方をみていきます。

そして最後に、今後データ分析業務をやっていく上で、必要となってくるであろう数学的な手法や、エンジニアリングツール等を紹介します。

数学的な手法では、実験計画法やMCMC、エンジニアリング面では、Linuxやクラウドサービスなどを簡単に紹介します。もちろん、ここですべてが網羅されているわけではありませんが、ビジネスの現場で使われている手法やツールですので、ぜひ参考にしてください（ただし、これらの手法やツールにとらわれることなく、課題に対する理解や最適なアプローチは何であるか、常に考えていくことが大事だということは忘れないでください）。

Special 2

深層学習を学ぶための準備

Keyword パーセプトロン、論理回路、ニューラルネットワーク、活性化関数、勾配（降下）法、バッチ学習、ミニバッチ学習、確率的勾配降下法、誤差逆伝播法、Chainer、Theano、Pytorch、Tensorflow


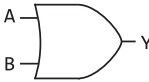
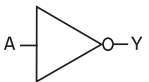


S-2- 1 パーセプトロン

まずは、ニューラルネットワークや深層学習の基礎となるパーセプトロンについて学びましょう。パーセプトロンは、複数の値を受け取って処理をし、1つの結果を返す関数です。

S-2- 1-1 AND論理回路の例

例として、ANDの論理回路（論理ゲート）の実装を考えます。AND関数は、たとえば0か1が入力されたとき、どちらも1の場合に1を返す関数です。以下の図S-2-1が参考になります。

AND関数の回路記号や真理表（入力された0と1の演算と出力を対応させた表）は一番上の行にあります。真理表の具体的な見方は、たとえば、AとBが両方とも1の場合に、アウトプットとなるYが1になるというのが、表からわかります。

論理演算の種類	論理式	記号回路	真理値表															
AND	$Y=A \cdot B$		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	0	0	1	0	1	0	0	1	1	1
A	B	Y																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR	$Y=A + B$		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	1
A	B	Y																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
NOT	$Y=\overline{A}$		<table><tr><th>A</th><th>Y</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	Y	0	1	1	0									
A	Y																	
0	1																	
1	0																	
NAND	$Y=\overline{A \cdot B}$		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	1	0	1	1	1	0	1	1	1	0
A	B	Y																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR	$Y=\overline{A + B}$		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	0
A	B	Y																
0	0	1																
0	1	0																
1	0	0																
1	1	0																

図S-2-1 論理演算の種類と真理表

S-2- 1-2 パーセプトロンでAND関数を作る

以下に示すのはAND関数をパーセプトロンとして構成した例です。ある閾値 (theta) と入力された値に対する重み (w1、w2) が設定されており、それらの入力値と重み付けの演算結果 (tmp) がその閾値を超えるかどうかで結果を判定しています。

入力

```
def and_func(x1,x2):
    w1,w2,theta = 0.5,0.5,0.7
    tmp = x1 * w1 + x2 * w2
    if tmp <= theta:
        return 0
    elif tmp > theta:
        return 1
```

上記の関数の実行結果は以下となります。たとえば、入力を0と1にしたときの出力は0、入力を1と1にしたときの出力は1となり、結果はAND演算のようにになっていることがわかります。

入力

```
print('0 かつ 0:',and_func(0,0))
print('0 かつ 1:',and_func(0,1))
print('1 かつ 0:',and_func(1,0))
print('1 かつ 1:',and_func(1,1))
```

出力

```
0 かつ 0: 0
0 かつ 1: 0
1 かつ 0: 0
1 かつ 1: 1
```

上記はとてもシンプルな実装で、特定の重みで入力値を0と1に限定すると先ほどのような論理ゲートを模倣できます。そして、複数のパーセプトロンをつなぎ合わせることで、ニューラルネットワーク(後述)を構築できます。また、図S-2-1をみていただくとわかる通り、他には、OR関数やNOT関数などもあります。これらの関数の実装などについて詳しく知りたい方は本書巻末の参考文献「A-29」をご覧ください。

Let's Try

OR関数はどのように実装しますか。考えて、実装してみましょう。

Practice

【練習問題S-1】

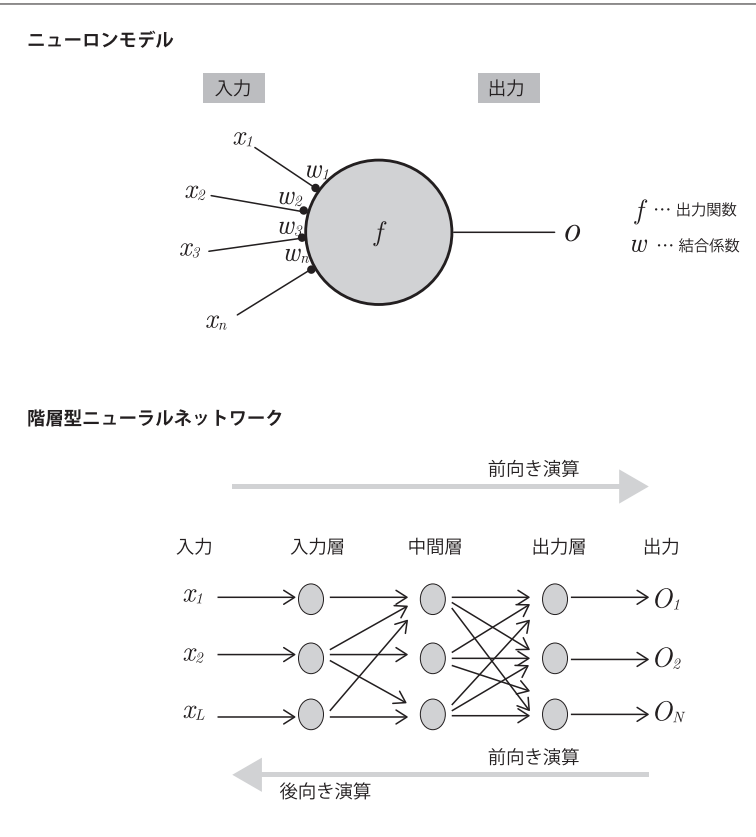
NAND (NOT AND) 関数を作成して、それが合っているかどうか確かめてください。

答えはTokutenAnswer.ipynb

S-2- 2 ニューラルネットワーク

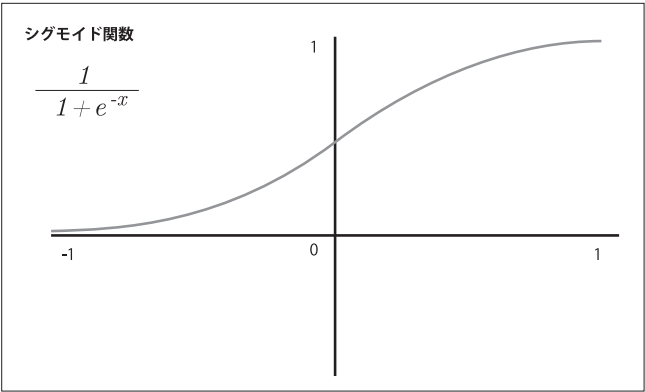
パーセプトロンを学んだら、ニューラルネットワークを学びましょう。
ニューラルネットワークは、順伝播型ネットワークやフォワードネットワークともいわれ、層状に並べたユニットが、隣り

合った層と結合しており、入力情報から出力情報に一方的に伝播していくモデルです。先ほどのパーセプトロンでは2つの入力値を使ってシンプルな条件式で判定して出力を決定していましたが、ニューラルネットワークでは活性化関数と言われる関数を使って計算して、出力を決定します。
以下の図S-2-2を参考にしてください。上側の「ニューロンモデル」では、入力値 x_1 から x_n が、 W_1 から W_n によって重みづけされ(掛け算され)、 f を関数として計算をして、その結果を出力します。下側の「階層型ニューラルネットワーク」は、層が複数になっているイメージです。



図S-2-2 ニューロンモデルと階層型ニューラルネットワーク
https://thinkit.co.jp/article/30/2/より引用(一部編集)

活性化関数には、シグモイド関数(図S-2-3参照)、正規化線形関数等が使われます。シグモイド関数は、出力が0から1の間に緩やかに変化する関数です。詳細は、参考文献「A-29」で紹介している『深層学習』などをみてください。



図S-2-3 シグモイド関数

S-2- 3 確率的勾配降下法と誤差逆伝播法

本書の11章では、予測した目的関数の値と実際の値のずれ、すなわち誤差をできる限り小さくするために、誤差関数に最小二乗法を適用するアプローチ等を学んできました。私たちがやりたいのは、この誤差関数を最小化する値（パラメータ）を求めることです。そのアプローチ方法をさらに詳しく紹介します。

S-2- 3-1 勾配法

誤差関数を最小化する値を探すとき、簡単な関数（二次関数の放物線など）だと求めやすいのですが、複雑な関数になると、どこで最小値を取るかを解析的に求めることが困難なケースがほとんどです。そのようなときに、まずはどこかに値をとって、値が小さくなる方向の勾配を求めて最小値を探していくのが勾配法です。これを繰り返し計算し、最小値がどこにあるのか探していきます。数式の傾きがその方向を決めるため、数学的には関数の偏微分を求めます。なお、最小値を求めるために下方向に向かって値を収束させることが多いので、勾配降下法というたりもします。

S-2- 3-2 確率的勾配降下法

さてデータを使って学習させるのに、訓練データをすべて使って一気に学習をさせる方法を**バッチ学習**といいます。従来はこのアルゴリズムが採用されていました。しかし、今日のデータ状況からみて、膨大なデータをすべて使って学習させるのには、かなりのコスト（時間やお金）がかかります。そこで、訓練データの中から無作為にデータを選んで、そのデータを使って学習をしていく（重み付けしていく）方法である**ミニバッチ学習**が使われています。このアプローチを取れば、大量なデータをすべて処理する必要がなくなり、計算コストを下げることができます。なお、データ1つ1つを取り出して学習させる**オンライン学習**もありますが、ここでは省略します。**確率的勾配降下法**は、このミニバッチ学習（訓練データのサンプルを1つまたは一部）を使って、勾配降下法でパラメータの更新をしていく法です。すべてのデータを使っていないため、計算効率が向上し、さらに局所的な解になってしまうリスクを抑えることができます。

S-2- 3-3 誤差逆伝播法

ただし、確率的勾配降下法であっても、改善できるとはいえ、計算時間がかかることもあります。それを解決する方法として、**誤差逆伝播法**があります。これは、重み付けのパラメータの勾配計算を効率良く行う手法です。逆伝播とは、出力層側から入力層に誤差情報を伝えていくことをいい、誤差逆伝播法はこのアプローチをとります。図S-2-2の下の図の、後ろ向き演算がそのイメージです。

以上で、概念的な紹介は終わりです。さらに学びたい方には、上で紹介した参考文献「A-29」の『ゼロから作るDeep Learnin』などがオススメです。

S-2- 4 深層学習のライブラリ

さて、これまでは深層学習の前学習ということで、パーセプトロンや確率的勾配降下法などを学びました。今後、深層学習を学ぶ方は、ChainerやTheano、Tensorflow、Pytorchという深層学習の計算を実行するためのライブラリを使っていくことになりますので、ここで少し紹介します。

Chainer

株式会社Preferred Networksが開発するディープラーニングのフレームワークです。いろいろなニューラルネットワークの構造に対応しており、GPUもサポートしています。詳細は、以下のサイトをご覧ください。
<http://chainer.org/>

Theano

モントリオール大学のBengio教授によって開発されている数値計算をするためのライブラリです。ディープラーニング自体の実装ではなく、それを計算するためのいろいろなサポート（微分計算、コンパイラ）などをしています。こちらもGPUで使うことができます（2017年11月の1.0リリースを以て開発が終了しています）。
<http://deeplearning.net/software/theano/>

Pytorch

Pytorchは2つの特徴をもったPythonのライブラリで、1つ目がGPUを使ったテンソル計算ができることと、2つ目が深層学習の計算ができるということです。深層学習のライブラリとしては後発ですが、人気がでているようです。なお、テンソルとはベクトルや行列をさらに一般化した概念で、多次元配列のデータを扱います。
<https://pytorch.org/>

Tensorflow

Googleがリリースした機械学習やディープラーニングのライブラリです。さまざまなOSに対応しています。Googleの社内での研究やサービスの開発において実際に使われており、公開後に利用者数が一気に増えました。
<https://www.tensorflow.org>

Keras

上記のTensorFlowなどのラッパーライブラリとして使われ、ディープラーニングの実装を簡易にできるのが特徴です。とても使いやすいので、初心者の方にもおすすめです。<https://keras.io/ja/>

以上で、深層学習を学ぶための準備を終わります。近年は深層学習に関する理論的な本や実装に関する良い本も出ています。参考書籍「A-30」をお勧めします。この講座を終えられた方ならスムーズに入れるのではないかと思います。また、深層学習については参考書籍「A-30」の『深層学習』がとても有名です。なお、英語版はネットで公開されています（参考URL「B-22」）。

Special 3

Pythonの高速化

Keyword 並行処理、並列処理、プロセス、スレッド、GIL、multiprocessing、JITコンパイラ、numba、Cython、コンパイル言語、インタープリタ言語、

ここからは、Pythonの処理を高速化するためのツールやアプローチについて学びます。Pythonを高速化するにはさまざまな方法がありますが、ここでは、並列処理をするmultiprocessing、コンパイラによるNumba、Cython等について説明します。

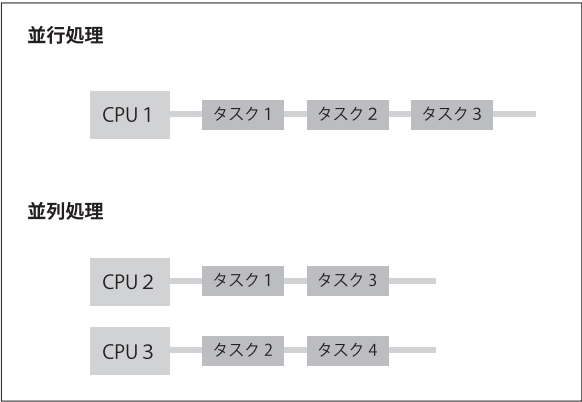
Pythonの高速化については、参考文献「A-31」があります。「A-31」の『ハイパフォーマンスPython』は、全体的なコンピューターシステム視点でPython処理のボトルネックとなる箇所を探すために、プロファイリング(システムをテストして遅い箇所を特定する)を実施したり、テクニカルな視点(リストとタプルの違いと扱い方など)でも説明がされています。『科学技術計算のためのPython入門』も高速化について、いくつかの章で解説がされていますので、参考にしてください。『エキスパートPythonプログラミング改訂2版』にも、最適化や並行処理について述べられています。ただし、Pythonによる高速化や処理の最適化は、まず動くコードを作成したあとに考えるのが原則です。システム全体として最適化をするためには、まずその全体的な動きをつかむことが重要で、局所的に最適化しても全体として最適化できるとは限りません。目的となるシステム、処理を作成した後、計算スピードに問題があるときに、検討するようにしましょう。また、やみくもに計算時間を計測するのではなく、どこに問題がありそうか仮説を立て、プロファイリングしていくことも重要です。

S-3- 1 並列処理

Python高速化のために、まずmultiprocessingのモジュールを使った処理をみてきます。その前に、周辺で必要となる概念(並列処理と並行処理、スレッドとプロセス、GILなど)について紹介します。ここでは、簡単な用語の説明に留めるので、詳細は先ほど紹介した参考文献等をみてください。

S-3- 1-1 並行処理と並列処理

まずは並行処理と並列処理の違いからです。並行処理と並列処理は同じような用語ですが、別の概念です。並行処理は複数のタスクを非同期で実行していきます。一方、並列処理は、並行処理の1つであり、複数のCPUを使って複数のタスクを同時に処理をしてきます。言葉の説明だけでは若干イメージがわきにくいので、図S-3-1をみてみましょう。上の図が並行処理で、下の図が並列処理です。並行処理では1つのCPUが2つのタスクを非同期で実行していますが、並列処理では、CPU1とCPU2の2つのCPUがそれぞれ同時に異なるタスクを処理しているのがわかります。



図S-3-1
並行処理と並列処理

複数のCPUコアを使うときに意識しなければいけないのが、グローバル・インタプリタ・ロック(GIL)です。これは、Pythonのプロセスはコア数に関わらず、一度に1つの命令しか実行しないことを言います。しかし以下で紹介するmultiprocessingの並列処理を使えば、プロセスとスレッドを使った並列処理を実現することができ、1つのマシンで複数のコアを使うことができます。なおプロセスとは、プログラムの実行単位のこと、スレッドとは、プロセスで作成される処理の単位のことをいいます。

S-3- 1-2 multiprocessingを使った並列処理の例

用語の説明はここまでにして、multiprocessingのサンプルコードを実際にみていきましょう。**なお、環境等によって、計算時間が変わってきますので、記載通りの結果にならないこともあります。ご了承ください。**以下のコードは、入力した値を2乗してそれを表示し、0.5秒待つ処理をするものです。このプログラムは何も特別な処理をせず、普通に実装しているだけのコードです(ただし、あくまで並列処理のイメージをもってもらうための実装で、これがシステムの最適であるとは限りません)。

入力

```
import time

# 入力値を二乗して結果を表示し、0.5秒待つ
def calc(x):
    a = x**2
    print(a)
    # 0.5秒待つ
    time.sleep(0.5)

# ここから処理を始める
if __name__ == '__main__':

    # 計算開始時間
    start_time = time.time()

    # リストデータの作成
    data = [t for t in range(0,10)]
```

出力

```
0
1
4
9
16
25
36
49
64
81
Calc Time:5.008[s]
```

```
# 関数の呼び出しとリスト化
[calc(x) for x in data]

# 計算時間の測定
print('Calc Time:{0:0.3f}[s]'.format(time.time() - start_time))
```

上記の実装では、著者の環境では計算時間は約5秒でした。

次は、multiprocessingを使って並列処理してみましょう。multiprocessing.Processなどを使って、次のように実装します。

入力

```
import time
import multiprocessing

# 入力されたデータから以下の calc を使ってリスト化する関数
def worker(data):
    [calc(x) for x in data]

# 入力値を二乗して結果を表示し、0.5秒待つ
def calc(x):
    a = x ** 2
    print(a)
    time.sleep(0.5)

# ここから処理開始
if __name__ == '__main__':

    start_time = time.time()

    # プロセスを分けるための設定
    split_data = [[0, 1, 2], [3, 4],[5, 6],[7, 8, 9]]

    jobs = []
    for data in split_data:
        job = multiprocessing.Process(target=worker, args=(data, ))
        jobs.append(job)
        job.start()

    [job.join() for job in jobs]

    print('Calc Time:{0:0.3f}[s]'.format(time.time() - start_time))
```

出力

```
0
9
25
49
1
16
36
64
4
81
Calc Time:1.536[s]
```

計算時間は約1.5秒に改善しました。

S-3- 1-3 マルチスレッドを使った例

次に、マルチスレッドを使った処理を見ていきましょう。以下の処理では3つのURLにリクエストを送って、待ち時間の合計を計算しています。こちらは普通の実装です。

入力

```
from urllib.request import urlopen
import time

current_time = time.time()
urls = ['http://www.google.com', 'http://www.yahoo.co.jp/', 'https://www.bing.com/']
for url in urls:
    response = urlopen(url)
    html = response.read()
    print('Calc Time:{0:0.3f}[s]'.format(time.time() - current_time))
```

出力

```
Calc Time:0.471[s]
```

上の結果から、約0.47秒かかったことがわかります。

次にこれをマルチスレッドを使って処理してみましょう。threadingを使います。

入力

```
from urllib.request import urlopen
import threading, time

def get_html(url):
    current_time = time.time()
    response = urlopen(url)
    html = response.read()
    print(url + ': ' + str(time.time() - current_time))

urls = ['http://www.google.com', 'http://www.yahoo.co.jp/', 'https://www.bing.com/']
threads = []

# Start Threads
current_time = time.time()
for url in urls:
    thread = threading.Thread(target=get_html, args=(url,))
    thread.start()
    threads.append(thread)

# Wait Threads
for thread in threads:
    thread.join()

print('Calc Time:{0:0.3f}[s]'.format(time.time() - current_time))
```

出力

```
http://www.google.com: 0.07345438003540039
http://www.yahoo.co.jp/: 0.1014854907989502
https://www.bing.com/: 0.21059107780456543
Calc Time:0.212[s]
```

処理時間は約0.2秒になり、先ほどより改善されているのがわかります。

S-3- 2 並行処理と並列処理

次は、JIT (Just in Time) コンパイラを使って、高速化しましょう。ここでは、LLVMベースのコンパイラnumbaを使います。

高速化するには、高速化したい関数の前にデコレータ（入力として関数を受け取り別の関数を返す）の@jitをつけます。そうすることでJITコンパイラが機械語にコンパイルしてくれて、Cにも匹敵する計算性能が出せるようになります。

以下に示すのは、複素数の計算をする例です。普通にPythonの機能として実装したmulti_abs_basicとNumpyで実装したmulti_abs_numpy、そして、先頭に「@jit」を付けて、multi_abs_basicと同じ処理をJIT化したmulti_abs_numbaの3つの関数を用意しました。

入力

```
from numba import jit
import numpy as np
import time

#それぞれの要素の掛け算を実行して、リストにする
#普通の実装
def mult_abs_basic(N,x,y):
    r = []
    for i in range(N):
        r.append(abs(x[i] * y[i]))
    return r

#numpyの実装
def mult_abs_numpy(x,y):
    return np.abs(x*y)

#JITの実装
@jit('f8[:](i8, c16[:], c16[:])',nopython=True)
def mult_abs_numba(N,x,y):
    r = np.zeros(N)
    for i in range(N):
        r[i] = abs(x[i] * y[i])
    return r
```

まずは、計算をするための変数の準備をします。以下のJは複素数を扱うために使っています。

入力

```
N = 1000000

x_np = (np.random.rand(N)-0.5) + 1J*(np.random.rand(N)-0.5)
y_np = (np.random.rand(N)-0.5) + 1J*(np.random.rand(N)-0.5)

x = list(x_np)
y = list(y_np)
```

それではまず、普通に計算したときの処理時間を見てみましょう。

入力

```
start_time = time.process_time()
b1 = mult_abs_basic(N,x,y)
print('Calc Time:{0:0.3f}[s]'.format(time.process_time()-start_time))
```

出力

```
Calc Time:0.284[s]
```

処理時間は約0.28秒でした。次は、numpyを使って計算した場合です。

入力

```
start_time = time.process_time()
b1 = mult_abs_numpy(x_np,y_np)
print('Calc Time:{0:0.3f}[s]'.format(time.process_time()-start_time))
```

出力

```
Calc Time:0.043[s]
```

計算時間は、約0.043秒でした。以前学んだ通り、numpyを使った方が計算時間が大幅に改善されているのがわかります。

最後にJITコンパイラを使って計算した時を見てみましょう。

入力

```
start_time = time.process_time()
b1 = mult_abs_numba(N,x_np,y_np)
print('Calc Time:{0:0.3f}[s]'.format(time.process_time()-start_time))
```

出力

```
Calc Time:0.029[s]
```

計算時間は0.029秒になっています。

Cで実装されているNumpyよりもかなり計算時間が削減されていることがわかります。

S-3- 3 Cython入門

Pythonの高速化の最後として、Cythonを使ったサンプルコードを紹介します。

PythonとCとの違いは、Pythonがインタプリタ言語に対して、Cはコンパイラ言語であるという点です。Pythonのようなインタプリタ言語はすぐに実行され、その処理結果の確認ができますが、C言語は実行前に機械語に直す過程（コンパイル）が必要であるため、その時間がかかります。

しかし、インタープリタ言語のデメリットとしては、実行速度そのものが遅くなってしまう点が挙げられます。コンパイラ言語は、事前に機械語に置き換えるため、実行の際の速度が速くなります。

ここで扱うCythonは、Pythonのように簡単に実装でき、コンパイルは必要になりますが、実行速度を早めることができます。

CythonはNumpyやScipyなどのさまざまなライブラリで使われており、本来は背景やその仕組み(CPythonとの関係性等)についても詳しく記載すべきですが、本書での目的は、まずは実装して動かすということに重きを置いているために、省略します。

S-3- 3-1 PythonとCythonの速度を比較する

ここでは、PythonとCythonの計算スピードの比較をします。まずは、Jupyter Notebookにて、以下のようにマジックコマンドを実行します。この%load_ext Cythonと以下で述べる%%cythonを実行することで、Jupyter Notebook内でもコンパイルが可能になります。

入力

```
%load_ext Cython
```

ここでは以前扱った素数とフィボナッチ数を計算するプログラムをPythonとCythonで実装し、それぞれの速度の違いを比較してみましょう。

S-3- 3-2 Pythonによる実装

まずは普通のPythonによる実装です。1つ目が以前にも扱ったフィボナッチ数列の生成関数です。

入力

```
# フィボナッチ数列の作成
def pyfib(n):
    a, b = 0.0, 1.0
    for i in range(n):
        a, b = a+b, a
    return a
```

2つ目は素数生成の関数です。

入力

```
import numpy as np

# 素数の作成
def pyprimes(kmax):
    p = np.zeros(1000)
    result = []

    # 最大個数は1000個
    if kmax > 1000:
        kmax = 1000
```

```
k = 0
n = 2
while k < kmax:
    i = 0
    while i < k and n % p[i] != 0:
        i += 1

    if i == k:
        p[k] = n
        k += 1
        result.append(n)
    n += 1
return result
```

S-3- 3-3 Cythonによる実装

次はCythonによる実装です。普通のPythonとの書き方の違いは、手前に%%cythonマジックコマンドを記載し、変数の前に型(整数型、倍精度浮動小数点型など)が必要になるという点です。以下に示すように、「cdef int」や「cdef double」などのように、cdefの後に型を書いて、その後ろに変数名を書きます。

入力

```
%%cython -n test_cython_code
# フィボナッチ数列生成 (Cythonバージョン)
def fib(int n):
    cdef int i
    cdef double a=0.0, b=1.0

    for i in range(n):
        a, b = a+b, a
    return a

# 素数生成 (Cythonバージョン)
def primes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    result = []

    if kmax > 1000:
        kmax = 1000

    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i += 1

        if i == k:
            p[k] = n
            k += 1
            result.append(n)
        n += 1
    return result
```


速度の違いを確認する

それでは、普通のPythonのコードとCythonのコードを比較してみましょう。まずは、フィボナッチ数列から計算します。

入力

```
# フィボナッチ計算比較

# 普通の Python で実行
%timeit pyfib(1000)

# Cython で実行
%timeit fib(1000)
```

出力

```
43 μs ± 20.6 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
1.25 μs ± 1.19 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

計算時間がなんと約35倍(43μsから1.25μs)も改善しています。ちなみに、1μsは100万分の1秒です。次は素数の計算です。

入力

```
# 素数計算比較

# 普通の Python で実行
%timeit pyprimes(1000)

# Cython で実行
%timeit primes(1000)
```

出力

```
272 ms ± 477 μs per loop (mean ± std. dev. of 7 runs, 1 loop each)
2.19 ms ± 1.61 μs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

こちらは100倍以上(272msから2.19ms)の改善となりました。ちなみに、1msは1000分の1秒です。Cythonについては参考文献「A-32」が参考になります。

以上で、Pythonの高速化は終わりです。他にも、Pythonで高速化するためのツールとして、分散処理をするDaskやBlazeなどがありますので、興味のある方は参考文献「A-33」や参考URL「B-23」などを調べてみてください。冒頭でも述べましたが、ここで紹介した実装が必ず最適というわけではありません。Pythonでの計算に時間がかかる時は、上記のような手法を用いて、高速化できるかどうか検討して試してみてください。

Special 4

Spark入門

Keyword RDD、key/value、mapreduce、sparkSQL

Sparkは分散処理をするためのツールでビッグデータを解析するためのソフトウェアの1つです。機械学習のライブラリも扱え、しかもSQLのように簡単に集計ができたり、リアルタイムに分析できます。SparkはScalaベースで作られていますが、Scala以外にもPythonやJavaなどからも利用可能です。本講座では、Pythonとの連携をメインとして、その処理方法の基礎を学びます。特にPysparkについて、その簡単な使い方を紹介します。

今までは、NumpyやScipy、Pandas、Scikit-learnなどを使って、データの加工処理から機械学習のモデリングを実施してきました。扱ってきたデータもそれほど大きくなく、せいぜい数ギガ程度の大きさでした。しかし、データ分析の現場では、数百ギガやテラバイト級のデータがあり、それを扱う場面も多々あり、先ほどのライブラリだけでは、データが読み込めない、計算が終わらない、困難というケースもあります。そこで、そのような大量のデータの読み込み、加工、機械学習まで一貫して実行するためのSparkを使うことで、ビッグデータを扱うことが可能となります。

S-4- 1 PySpark入門

ここでは、PythonとSparkを連携させたPySparkの使い方について、基礎の基礎を学びましょう。PySparkはPythonからSparkを操作できるAPIです。

S-4- 1-1 Pysparkを使うための準備

まずは、PySparkを使うために、以下のコードを実行しましょう。PySparkを実行するための環境構築はAppendixをご覧ください。

入力

```
import findspark
findspark.init()

import pyspark
sc = pyspark.SparkContext(appName='myAppName')
```

以下のようにscコマンドを入力するとバージョン番号が表示されるので、spark(pyspark)が使えるようになったことを確認できます。

入力

```
sc
```

出力

```
SparkContext

Spark UI

Version
v2.2.0
Master
local [*]
AppName
myAppName
```

S-4- 1-2 データの読み込み

ここから、分析対象のデータの読み込みをします。sc.textFileの後にデータのある場所を指定しています。なお、ファイル名にアスタリスク(*)をつけて任意の文字を含むファイル名をまとめて読みこむことが可能です。同じような規則性のあるファイル名を扱う時は便利です。

3章で扱った「student-mat.csv」と「student-por.csv」を同時に読み込んでみましょう。ただし、ここでは重複等は気にせず、そのまま読み込みます。なお、該当のデータがある場所は絶対パスで指定してください。

入力

```
merge_student_data = sc.textFile('/root/userspace/data/student*.csv')
```

上記ではRDD (Resilient Distributed Dataset) としてデータをロードさせています。日本語では、「不変・並列実行可能な(分割された)コレクション」という意味です。RDDをベースにSparkはデータを分散処理させます。Sparkでは、このようなRDDの生成、既存のRDDの変換、結果を集計などをするためにRDDに対する呼び出しをしています。この段階ではまだ集計等はしていません。

次は、行数を数えています。ここで集計を開始します。

入力

```
# Line count
merge_student_data.count()
```

出力

```
1046
```

以下でデータの始めの行を表示します。RDDの後にfirst()をつけて実行します。

入力

```
merge_student_data.first()
```

出力

```
'school;sex;age;address;famsize;Pstatus;Medu;Fedu;Mjob;Fjob;reason;guardian;traveltime;studytime;failure
s;schoolsup;famsup;paid;activities;nursery;higher;internet;romantic;famrel;freetime;goout;Dalc;Walc;hea
lth;absences;G1;G2;G3'
```

take(5)で5行を抽出します。

入力

```
merge_student_data.take(5)
```

出力

```
['school;sex;age;address;famsize;Pstatus;Medu;Fedu;Mjob;Fjob;reason;guardian;traveltime;studytime;failur
es;schoolsup;famsup;paid;activities;nursery;higher;internet;romantic;famrel;freetime;goout;Dalc;Walc;hea
lth;absences;G1;G2;G3',
 'GP";"F";18;"U";"GT3";"A";4;4;"at_home";"teacher";"course";"mother";2;2;0;"yes";"no";"no";"no";"yes";"y
es";"no";"no";4;3;4;1;1;3;6;"5";"6";6',
 'GP";"F";17;"U";"GT3";"T";1;1;"at_home";"other";"course";"father";1;2;0;"no";"yes";"no";"no";"no";"yes"
;"yes";"no";5;3;3;1;1;3;4;"5";"5";6',
 'GP";"F";15;"U";"LE3";"T";1;1;"at_home";"other";"other";"mother";1;2;3;"yes";"no";"yes";"no";"yes";"yes
";"yes";"no";4;3;2;2;3;3;10;"7";"8";10',
 'GP";"F";15;"U";"GT3";"T";4;2;"health";"services";"home";"mother";1;3;0;"no";"yes";"yes";"yes";"yes";"y
es";"yes";"yes";3;2;2;1;1;5;2;"15";"14";15']
```

次は、lambdaも組み合わせて、フィルターをかけます。具体的には、GPという文字が含まれている行を抽出します。

入力

```
gp_lines = merge_student_data.filter(lambda line: 'GP' in line)
```

補足ですが、まだ上のフィルターの段階では、計算はしておらず、上記のcountやfirstを実行した時に計算がされます。これはSparkの、「実際に値が必要になるまでは計算をしない」という遅延評価によるものです。先ほどの、take()も実施した時に、計算がされます。Sparkではこの考え方がとても重要なので、押さえておいてください。

以下は行数をカウントしています。ここで、計算が実行されます。

入力

```
gp_lines.count()
```

出力

```
772
```

上記の結果より、GPが含まれる行数は772であることがわかりました。

また、カラム名が入っている行をカウントしましょう。先ほど2ファイルをそのまま読み込んでいるので、カラム名が入っている行は2行あるはずです。以下は、カラム名の1つであるschoolが含まれている行数をカウントしています。

入力

```
head_lines = merge_student_data.filter(lambda line: 'school' in line)
head_lines.count()
```

出力

```
2
```

これを応用して、MapReduceの処理を見てみましょう。以下では、それぞれのセルに入っているデータ(数字も)を1つの単語と見なし、それぞれいくつあるのかをカウントする処理をしています。

まず、splitを使って「;」で文字を分けています。

入力

```
split_words = merge_student_data.flatMap(lambda line:line.split(';'))
```

次に、mapで単語1つにカウント1を対応させ、reduceの処理でそれぞれ同じ単語のカウントを足しあげています。これがMapReduceの処理です。キーとなっているのが単語 (word) です。

入力

```
word_counts = split_words.map(lambda word:(word,1)).reduceByKey(lambda a,b:a+b)
```

collect()でRDD全体を取り出しますので、大きなデータを扱う際は注意してください。

入力

```
result = word_counts.collect()
```

返ってくる値はリスト型です。数が多いので表示結果は絞っています。

入力

```
result[0:10]
```

出力

```
[('Fedu', 2),
 ('Fjob', 2),
 ('schoolsup', 2),
 ('activities', 2),
 ('nursery', 2),
 ('higher', 2),
 ('internet', 2),
 ('romantic', 2),
 ('G1', 2),
 ('G3', 2)]
```

また、Statisticsライブラリ等を使うことで、基本統計量の計算も可能です。

入力

```
from pyspark import SparkContext
from pyspark.mllib.linalg import Vectors
from pyspark.mllib.stat import Statistics
import numpy as np
```

まずは、統計量を計算するためのデータを準備します。

入力

```
rdd = sc.parallelize([Vectors.dense([2, 0, 0, -2]),Vectors.dense([4, 5, 0, 3]),Vectors.dense([6, 7, 0, 8])])
```

次に、Statistics.colStatsを使って、基本統計量等を計算します。

入力

```
summary = Statistics.colStats(rdd)
```

以下は、順番にベクトルの各要素の平均、分散、ゼロでない数のカウントを計算しています。

入力

```
print (summary.mean())
print (summary.variance())
print (summary.numNonzeros())
```

出力

```
[ 4.  4.  0.  3.]
[ 4. 13.  0. 25.]
[ 3.  2.  0.  3.]
```

以上で、データの読み込みと簡単な集計の紹介は終わります。次は、SparkSQLを見ていきます。

Practice

【練習問題S-2】

S-4-1「PySpark入門」で使用したデータ「merge_student_data」に対して、schoolsupを含むレコードがどれだけあるかカウントしてください。

答えはTokutenAnswer.ipynb

S-4-2 SparkSQL入門

Sparkは、データベース (DB)のデータを操作するSQLも扱うことができます。以下は、そのモジュールの読み込みと準備をしています。なお、以下は、DBの基礎やSQLの基礎がある方を前提に説明をしていますので、あらかじめご了承ください。

入力

```
from pyspark.sql import SQLContext
from pyspark.sql.types import StructField,StringType,IntegerType,StructType,FloatType
sqlContext = SQLContext(sc)
```

3章で扱ったデータを対象に、SQL (ここではSparkSQL) を使って集計してみましょう。
データを読み込みます。RDDとしてロードしています。なお、該当のデータがある場所は絶対パスで指定してください。

入力

```
student_mat_data = sc.textFile('/root/userspace/data/student-mat.csv')
```

見出し列だけ読み込んで確認してみましょう。

入力

```
header = student_mat_data.first()
header
```

出力

```
'school;sex;age;address;famsize;Pstatus;Medu;Fedu;Mjob;Fjob;reason;guardian;traveltime;studytime;failure
s;schoolsup;famsup;paid;activities;nursery;higher;internet;romantic;famrel;freetime;goout;Dalc;Walc;hea
th;absences;G1;G2;G3'
```

このデータの区切り文字は特殊でした。以下のように ; を置き換えましょう。replaceを使います。

入力

```
schemaString = header.replace(';','')
schemaString
```

出力

```
'school,sex,age,address,famsize,Pstatus,Medu,Fedu,Mjob,Fjob,reason,guardian,traveltime,studytime,failure
s,schoolsup,famsup,paid,activities,nursery,higher,internet,romantic,famrel,freetime,goout,Dalc,Walc,hea
th,absences,G1,G2,G3'
```

さて、データベースのテーブルを準備するとき、フィールド名や型をすべて指定しなければいけませんが、30個以上もあるので、設定するのは大変です。少し乱暴ですが、以下のようにプログラムを書いて、作業を効率化しましょう。取り急ぎすべて文字型にしています。個別に後から変更することも可能です。

入力

```
fields = [StructField(field_name, StringType(), True) for field_name in schemaString.split(',')]

これをデータ構造としてスキーマを定義します。
```

入力

```
schema = StructType(fields)
```

先ほど確認した見出し列を除いたデータを準備します。

入力

```
Header = student_mat_data.filter(lambda l: "school" in l)
NoHeader = student_mat_data.subtract(Header)
NoHeader.count()
```

出力

```
395
```

以下は、見出し列を除いたデータの1行目です。

入力

```
NoHeader.first()
```

出力

```
"GP";"F";15;"U";"LE3";"T";1;1;"at_home";"other";"other";"mother";1;2;3;"yes";"no";"yes";"no";"yes";"yes";"yes";"no";4;3;2;2;3;3;10;"7";"8";10'
```

また次に、文字が”で囲まれているため、map関数とlambda関数を使って、その文字を消去しています。

入力

```
NoHeader2 = NoHeader.map(lambda l: l.replace('\",''))
NoHeader2.first()
```

出力

```
'GP;F;15;U;LE3;T;1;1;at_home;other;other;mother;1;2;3;yes;no;yes;no;yes;yes;yes;no;4;3;2;2;3;3;10;7;8;10'
```

さらに、; で文字が区切られているため、; で分割します。

入力

```
NoHeader3 = NoHeader2.map(lambda l: l.split(';'))
```

次は、上のデータをデータフレームにします。先ほどのスキーマで設定します。sqlContext.createDataFrameを使います。

入力

```
data_frame = sqlContext.createDataFrame(NoHeader3, schema)
```

最後に、SQLを使うために、上のデータフレームをテーブルにして、登録します。

入力

```
print(data_frame.printSchema())
```

出力

```
root
 |-- school: string (nullable = true)
 |-- sex: string (nullable = true)
 |-- age: string (nullable = true)
 |-- address: string (nullable = true)
 |-- famsize: string (nullable = true)
 (以下省略)
```

それでは、SQLのselect文を使ってみましょう。基本的にsqlContext.sqlの中でSQLを記述すれば、大丈夫です。なお、集計結果としては、前半のChapterで実行した結果と同じになります。

入力

```
count_result = sqlContext.sql('select sex,avg(age) as avgAge from tmp_table group by sex')
print(count_result.show())
```

出力

```
+---+-----+
|sex|          avgAge|
+---+-----+
|  F| 16.73076923076923|
|  M|16.657754010695186|
+---+-----+

None
```

以上で、Sparkの説明は終わりになります。Sparkではさらに、前に学んだ機械学習の手法もMLibを使ってRDDに適応させることが可能です。

 今回は簡単な実装の紹介のみで、一般的な環境ではSparkの凄さを実感できませんが、将来的にAmazon Web Services (AWS、Amazonが提供するクラウドサービス) などてたくさんのサーバーが使えるときや分散処理を実施する時に、ぜひ使ってみてください。ちなみに、Sparkはインストール等が少し大変だと思うので、AWSが提供するAmazon Elastic MapReduce (EMR) が便利です。ノード数の指定などが簡単に設定できるので、はじめて使う方にはEMRはオススメです。さらに、あとで紹介しますが、Amazon Sage Markerを使えば、すぐにSparkをJupyter上で使えるので、これもオススメです。

 またSparkはScalaベースで作られており、色々な面でScalaからの方が扱いやすいと思いますので、本格的に使われる場合はScalaもプログラミング言語の選択肢として考えるのも良いと思います (PySparkは制約があります)。

 なお、参考文献「A-34」もご覧ください。

Practice

【練習問題S-3】

SparkSQLを使って、先ほど作ったテーブルに対して、school× sexを軸にして、それぞれのレコード数と、それぞれの平均年齢を表示するようにしてください。

答えはTokutenAnswer.ipynb

Special 5

その他の数学的手法とエンジニアリングツール

Keyword MCMC、階層ベイズ、実験計画法、生存解析、確率過程とランダムウォーク、時系列解析、トピックモデル、Linux、AWS、Hadoop、FPGA、Scala、R、Java、OpenCV、Tableau、PowerBI

今までは主にPythonを使ってきましたが、分析に関連するソフトウェアや分析に役立つツールを、まばらではありますが紹介します。この講座のみですべてを学ぶことはできませんが、将来的なアンテナを張れるように単語を知っておくだけでもためになると思います。ここは、データサイエンティストになるために必要なスキルの2つ目エンジニアリング力を磨くためのコンテンツ紹介になります。

Linux (コマンドライン、シェル)

OSの1つです。将来、分析をするだけではなく、分析をするための環境構築等にも必要になってくるスキルです。初学者にはなかなかハードルが高いですが、Linuxを徐々に使い慣れていけば、分析環境を構築したり、実際に分析するのに、色々と選択肢が広がるでしょう。

Hadoop

分散処理システムです。よく取り上げられる簡単な例が、ある文章の中に出てくる単語をそれぞれカウントするときはどうやって効率よく数えていくのかという問題です。

 処理としては、文章の中にある単語に数字を対応させ (Map処理)、後から集めて集計する (Reduce処理) という処理をHadoopがやります (分散処理と監視など)。単語のカウントは、上記のSparkの例でも扱いました。また、Sparkは、このHadoopと合わせて使われることも多いです。

 ビックデータ分析といえばHadoopというくらい、大量のデータを処理するのによく紹介されます。ただ、Hadoopはバッチ処理になりますので、リアルタイムに分析をする場合には、Spark等と組み合わせる必要があります。

FPGA

昨今、深層学習計算などでたくさんのCPUやGPUが使われており、ソフトウェアの面からだけではなく、ハード面で実装をすることが求められています。

 このFPGAはプログラム可能なICで、Field Programmable Gate Arrayの略です。論理仕様をプログラムすることができ、ユーザーの思い通りの論理回路を実現できる論理デバイスです。メリットとしては、CPUが処理する段階でロジックを組むことができるので、処理スピードがかなり改善されます。

 ハードウェア記述言語のVHDLなどで書かれており、始めるのにかなりハードルが高いツールですが、最近はPynqなどPythonからも実装ができるようになってきているようです。FPGAは、データ分析中級者向けというより、レベル的にはかなりの上級者 (トップクラスのデータサイエンティストかエンジニア) 向けだと思いますが、本書を読了した人の中から、データ分析の第一線で活躍している人が出てくることを期待して、紹介をしました。

 インフラよりの話ではありますが、分析するのにCPUレベルで考えなければいけない時代が来るのかもしれない。

用分野としては、MicrosoftのBingの検索エンジンやゲノム科学解析、ゲームユーザーの振り分け、金融の高頻度取引などに使われており、いろいろな企業が注目しています。ただこれも用途やデータに応じて使うものですので、FPGA（やGPUなど）を使えば必ず速くなるというものではないので注意しましょう。

AWS (Amazon Web Services)

Amazonのクラウドサービスです。サーバー構築やデータストレージ環境などさまざまなサービスが提供されています。あるイベントが起きたときにメールを通知することができるなど、機能も多種多様です。クラウドサービスを提供する会社の中では今のところシェアが圧倒的に高いです。試験的に何か始める時には、コストを抑えることができるため便利です。また、一時的に大量のサーバーを立ち上げるなど、分散処理システムを構築したい場合に使いやすいです。HadoopやSparkがあらかじめインストールされるAmazon EMRや、データベースを分散処理により高速化させるRedshiftなど、分析で使われることも多いです。AWSについては、最近Amazon SageMakerというデータサイエンティストや機械学習エンジニア向けのサービスが提供されており、機械学習モデルの構築やトレーニング、検証などが簡単に実行できます。AWSにログインして、Jupyter環境もすぐに利用でき、さらにTensorFlowやPyTorchなど深層学習で使うライブラリもあらかじめロードされています。また、Sparkなどもすぐに利用できて、使うマシン（インスタンス）の数なども指定でき、ハイパーパラメータチューニングなども高速に実行することもできますし、データストレージ環境のS3にデータを貯めておけば、すぐに連携することができて、オススメです。

その他、Jupyter環境がすぐに利用できるサービスとして、冒頭でも紹介したGoogle Colabolatoryがあります。以前は、分析をするための環境を用意するのに時間がかかったのですが、これらのサービスを使えば、あっというまに分析できる環境を手に入れることができ、時間の節約ができます。ただし、これらのサービスはインターネットが繋がっていないと利用できないのと、AWSについてはお金がかかったりするので、節約したい方はローカルの環境でANACONDAをインストールして、分析環境を構築するとよいでしょう。

R統計ソフトウェアです。統計的な手法のライブラリが豊富です。変数選択法など、Pythonのパッケージではできないものも、いくつかあります。もしこれらのパッケージをPythonで利用したい場合は、PythonからRのスクリプトも呼び出すことができます。

可視化ツール

Tableau、PowerBIなどがあります。近年はインフォグラフィックスというデータの可視化が注目されており、これらのツールを使うことで、簡単にデータを可視化することができます。

OpenCV

画像を処理するためのライブラリです。人の顔を認識したりすることもできます。Pythonと連携が可能で、Jupyter Notebookと組み合わせて実行すると便利です。

ここで紹介したツールやプログラミング言語の他にも、Java、Rudy、PHP、Scalaや機械学習や統計分析向けのJulia、Jumpも分析用のツールとしてよく使用されているようです。クラウドサービスとしてはAWSの他に、GoogleのGCPやIBMのBluemix、MicrosoftのAzureなどがあります。商用の統計解析ツールとしては、SAS、SPSS、Matlabなどがありますが、高額なので一般で買うのは難しいかもしれません。ただ、このような統計ソフトウェアの機能もPythonのライブラリで提供されているものが多くなってきましたので、PythonやRでもほとんど実現可能です。Pythonにはない統計手法を使いたい場合や、これらのソフトウェアに興味のあ

る方、大学や会社で使う必要があるという方は、無償版がありますので、それらをインストールしたり、本などをみて勉強してください。参考文献を「A-36」にまとめました。

他、Pythonのライブラリ関連で、Excelを扱うためのopenpyxlや、昨今流行りのRPAツールを作成するためのpyautoguiなどもあります。これらを使うことで、社内にあるExcelのファイル処理やそのための自動化ツールを作成できるでしょう。興味のある方はこれらのキーワードで調べてみてください。

以上で、この章は終わります。お疲れ様でした。

Practice

Special Contents 総合問題

【総合問題S-1 深層学習の用語】
深層学習に関する以下の用語について、それぞれの役割やその意味について述べてください。また、ネットや参考文献等も使って調べてみてください。

- ・パーセプトロン
- ・ニューラルネットワーク
- ・勾配法
- ・バッチ学習
- ・ミニバッチ学習
- ・確率的勾配降下法
- ・誤差逆伝播法
- ・Chainer、Theano、TensorFlow

【総合問題S-2 Pythonの高速化とSparkに関する用語】
Pythonの高速化とSparkに関する用語について、それぞれの役割や意味について述べてください。また、ネットや参考文献等も使って調べてみてください。

- ・並列処理
- ・JITコンパイル
- ・Cython
- ・Spark
- ・RDD
- ・PySpark
- ・SparkSQL

※本総合問題の解答については省略します。本書の該当部分を再度読み直したり、インターネットで調べてみてください。

Special 6

教師なし学習で扱ったPCAのサンプルデータの相関係数が高い理由について

途中で使用するため、あらかじめ読み込んでおいてください。

入力

```
# データ加工・処理・分析ライブラリ
import numpy as np
import numpy.random as random
import scipy as sp
from pandas import Series, DataFrame
import pandas as pd
# 可視化ライブラリ
import matplotlib.pyplot as plt
import matplotlib as mpl
import seaborn as sns
%matplotlib inline
# 機械学習ライブラリ
import sklearn
# 小数第3まで表示
%precision 3
```

出力

```
'%.3f'
```

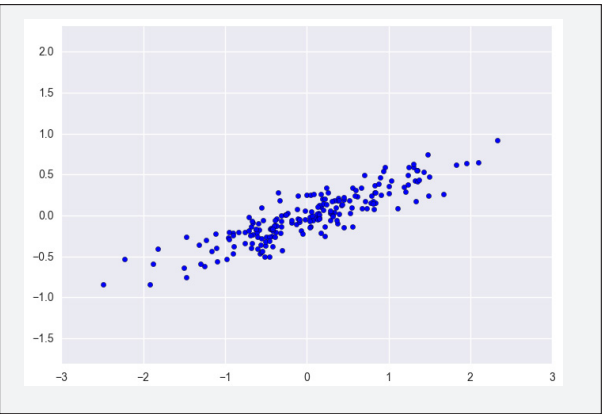
「9-3-1 主成分分析を試す」で、乱数を利用して2変数の200個のサンプルデータ X を得ましたが、 X の標本相関係数を計算してみましょう。

入力

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

rng_data = np.random.RandomState(1)
X = np.dot(rng_data.rand(2, 2), rng_data.randn(2, 200)).T
plt.scatter(X[:, 0], X[:, 1])
plt.axis('equal');
```

出力



入力

```
# 標本相関係数
np.corrcoef(X[:, 0], X[:, 1])
```

出力

```
array([[ 1. , 0.889],
       [ 0.889, 1. ]])
```

X の標本相関係数は0.889と高い値となりました。これは偶然なのでしょうか？ここではこの結果について、理論的な背景を理解するため、色々と考察してみましょう。まず、 X を10個求めてそれぞれの標本相関係数を求めてみます。

入力

```
for i in range(10):
    rng_data = np.random.RandomState(i)
    X = np.dot(rng_data.rand(2, 2), rng_data.randn(2, 200)).T
    print(np.corrcoef(X[:, 0], X[:, 1])[0,1])
```

出力

```
0.985748860832
0.889389954001
0.823512814441
0.990584244968
0.993175443077
0.999594882325
0.952192752595
0.899364799874
0.95599797861
0.0802633853592
```

値はばらついていますが、9個が0.8以上です。何か理由がありそうです。次に、標本相関係数を100000回計算してその平均値を求めてみます。

入力

```

corrcoef_sum = 0
n = 100000
for i in range(n):
    rng_data = np.random.RandomState(i)
    X = np.dot(rng_data.rand(2, 2), rng_data.randn(2, 200)).T
    corrcoef_sum += np.corrcoef(X[:, 0], X[:, 1])[0, 1]
corrcoef_ave = corrcoef_sum / n
print(corrcoef_ave)

```

出力

```
0.838868995091
```

0.839という高い値となりました。この挙動を確認してみましょう。

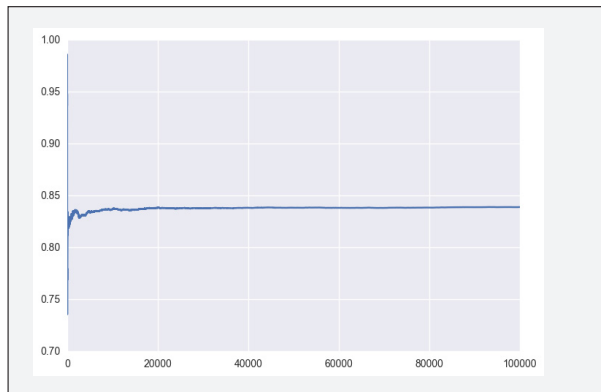
入力

```

corrcoef_sum = 0
corrcoef_ave = []
n = 100000
for i in range(n):
    rng_data = np.random.RandomState(i)
    X = np.dot(rng_data.rand(2, 2), rng_data.randn(2, 200)).T
    corrcoef_sum += np.corrcoef(X[:, 0], X[:, 1])[0, 1]
corrcoef_ave.append(corrcoef_sum / (i + 1))
plt.plot(corrcoef_ave)

```

出力



10000回程度で0.839近くに収束していることが、確認できます。この平均値をとる操作を5回行ってみましょう。

入力

```

m = 5
n = 10000
for j in range(m):
    rng_data = np.random.RandomState(j)
    corr_sum = 0
    corr_ave = []
    for i in range(n):
        X = np.dot(rng_data.rand(2, 2), rng_data.randn(2, 200)).T

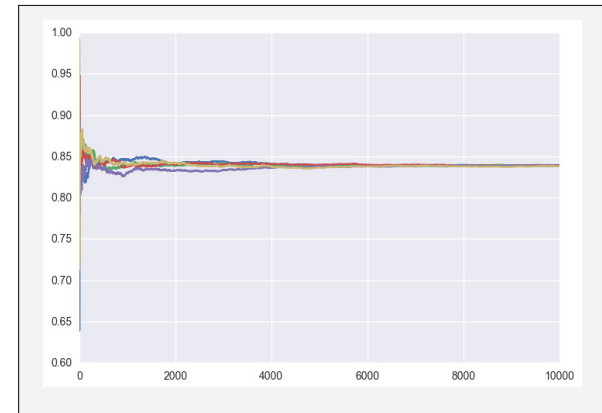
```

```

        corr_sum += np.corrcoef(X[:, 0], X[:, 1])[0, 1]
        corr_ave.append(corr_sum / (i + 1))
    plt.plot(corr_ave)

```

出力



5回とも0.839近くに収束していることが確認できます。この0.839という数字はどこから来ているのでしょうか?その理由を以下で考察します。

今回の X は

$$A = \begin{pmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \quad B = (\mathbf{b}_1, \dots, \mathbf{b}_{200}) = \begin{pmatrix} b_{11} & \cdots & b_{1\ 200} \\ b_{21} & \cdots & b_{2\ 200} \end{pmatrix}$$

$$X = \begin{pmatrix} x_{11} & x_{12} \\ \vdots & \vdots \\ x_{200\ 1} & x_{200\ 2} \end{pmatrix} = AB = (A\mathbf{b}_1, \dots, A\mathbf{b}_{200}) = \begin{pmatrix} \mathbf{a}_1\mathbf{b}_1 & \cdots & \mathbf{a}_1\mathbf{b}_{200} \\ \mathbf{a}_2\mathbf{b}_1 & \cdots & \mathbf{a}_2\mathbf{b}_{200} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & \cdots & a_{11}b_{1\ 200} + a_{12}b_{2\ 200} \\ a_{21}b_{11} + a_{22}b_{21} & \cdots & a_{21}b_{1\ 200} + a_{22}b_{2\ 200} \end{pmatrix}$$

という形で表されます。ただし、 A の成分は $[0, 1]$ 上の一様分布に従う独立な変数、 B の成分は標準正規分布に従う独立な変数、 A と B は独立とします。 X は X の転置行列を表します。まず、行列 A の成分を固定し、 B のみを乱数とみなして考えます。このとき、ベクトル $A\mathbf{b}_i$ は平均が0、共分散行列が $A^T A$ の2次元正規分布に従うことが知られています。ここで、 $A^T A$ は

$$A^T A = \begin{pmatrix} a_{11}^2 + a_{12}^2 & a_{11}a_{21} + a_{12}a_{22} \\ a_{11}a_{21} + a_{12}a_{22} & a_{21}^2 + a_{22}^2 \end{pmatrix}$$

となり、ベクトル $A\mathbf{b}_i$ の第一成分 $\mathbf{a}_1\mathbf{b}_i = a_{11}b_{1i} + a_{12}b_{2i}$ と第二成分 $\mathbf{a}_2\mathbf{b}_i = a_{21}b_{1i} + a_{22}b_{2i}$ の相関係数は

$$\frac{a_{11}a_{21} + a_{12}a_{22}}{\sqrt{(a_{11}^2 + a_{12}^2)(a_{21}^2 + a_{22}^2)}} = \frac{(\mathbf{a}_1, \mathbf{a}_2)}{\|\mathbf{a}_1\| \|\mathbf{a}_2\|} = \cos \theta$$

と表されます。ここで、 $(\mathbf{a}_1, \mathbf{a}_2)$ は \mathbf{a}_1 、 \mathbf{a}_2 の内積、 $\|\mathbf{a}_1\|$ 、 $\|\mathbf{a}_2\|$ はそれぞれの絶対値、 θ は \mathbf{a}_1 、 \mathbf{a}_2 のなす角度を表します。以上より、 A を固定した場合、 X は平均が0、相関係数が $\cos \theta$ の2次元正規分布からの独立な200個の標本とみなせます。

これまでは A を固定して考えてきましたが、 A を一様乱数としてとってきた場合、 $\cos \theta$ はどのような値をとるのでしょうか? それは、 $\cos \theta$ の期待値

$$\int_0^1 \int_0^1 \int_0^1 \int_0^1 \frac{a_{11} a_{21} + a_{12} a_{22}}{\sqrt{(a_{11}^2 + a_{12}^2)(a_{21}^2 + a_{22}^2)}} da_{11} da_{21} da_{12} da_{22}$$

を求めることでわかります。問題はこの積分をどのように求めるかですが、1つの方法として極座標変換があります。次の極座標変換を考えます。

$$(a_{11}, a_{12}) \rightarrow (r_1, \theta_1), (a_{21}, a_{22}) \rightarrow (r_2, \theta_2), a_{11} = r_1 \cos \theta_1, a_{12} = r_1 \sin \theta_1, a_{21} = r_2 \cos \theta_2, a_{22} = r_2 \sin \theta_2$$

ただし、元の積分範囲が[0,1]×[0,1]の正方形上であるため、変換した積分範囲に注意が必要です。今回は $\theta_1 = \frac{\pi}{4}$ 、 $\theta_2 = \frac{\pi}{4}$ の対角線で正方形を2つにわけます。

$$\begin{aligned} [0, 1] \times [0, 1] &\rightarrow D_1 \cup D_2, [0, 1] \times [0, 1] \rightarrow E_1 \cup E_2 \\ D_1 &= \{(r_1, \theta_1) \mid 0 \leq r_1 \leq \frac{1}{\cos \theta_1}, 0 \leq \theta_1 \leq \frac{\pi}{4}\}, D_2 = \{(r_1, \theta_1) \mid 0 \leq r_1 \leq \frac{1}{\sin \theta_1}, \frac{\pi}{4} < \theta_1 \leq \frac{\pi}{2}\} \\ E_1 &= \{(r_2, \theta_2) \mid 0 \leq r_1 \leq \frac{1}{\cos \theta_2}, 0 \leq \theta_2 \leq \frac{\pi}{4}\}, E_2 = \{(r_2, \theta_2) \mid 0 \leq r_1 \leq \frac{1}{\sin \theta_2}, \frac{\pi}{4} < \theta_2 \leq \frac{\pi}{2}\} \end{aligned}$$

この変換の元で、上の積分は次の形になります。

$$\int_0^1 \int_0^1 \int_0^1 \int_0^1 \frac{a_{11} a_{21} + a_{12} a_{22}}{\sqrt{(a_{11}^2 + a_{12}^2)(a_{21}^2 + a_{22}^2)}} da_{11} da_{21} da_{12} da_{22} = \iint_{E_1 \cup E_2} \iint_{D_1 \cup D_2} r_1 r_2 \cos(\theta_1 - \theta_2) dr_1 d\theta_1 dr_2 d\theta_2$$

積分範囲を以下のように4分割します。

$$(D_1 \cup D_2) \times (E_1 \cup E_2) = \bigcup_{i=1}^2 \bigcup_{j=1}^2 (D_i \times E_j)$$

そして、それぞれの範囲で積分をします。試しに、 $D_1 \times E_1$ での積分を求めます。

$$\begin{aligned} \iint_{E_1} \iint_{D_1} r_1 r_2 \cos(\theta_1 - \theta_2) dr_1 d\theta_1 dr_2 d\theta_2 &= \int_0^{\frac{\pi}{4}} \int_0^{\frac{1}{\cos \theta_2}} \int_0^{\frac{\pi}{4}} \int_0^{\frac{1}{\cos \theta_1}} r_1 r_2 \cos(\theta_1 - \theta_2) dr_1 d\theta_1 dr_2 d\theta_2 \\ &= \int_0^{\frac{\pi}{4}} \int_0^{\frac{\pi}{4}} \int_0^{\frac{1}{\cos \theta_2}} \int_0^{\frac{1}{\cos \theta_1}} r_1 r_2 \cos(\theta_1 - \theta_2) dr_1 dr_2 d\theta_1 d\theta_2 \\ &= \int_0^{\frac{\pi}{4}} \int_0^{\frac{\pi}{4}} \frac{\cos(\theta_1 - \theta_2)}{4 \cos^2 \theta_1 \cos^2 \theta_2} d\theta_1 d\theta_2 \\ &= \int_0^{\frac{\pi}{4}} \int_0^{\frac{\pi}{4}} \left(\frac{1}{4 \cos \theta_1 \cos \theta_2} + \frac{\sin \theta_1 \sin \theta_2}{4 \cos^2 \theta_1 \cos^2 \theta_2} \right) d\theta_1 d\theta_2 \\ &= \frac{1}{4} \int_0^{\frac{\pi}{4}} \frac{d\theta_1}{\cos \theta_1} \int_0^{\frac{\pi}{4}} \frac{d\theta_2}{\cos \theta_2} + \frac{1}{4} \int_0^{\frac{\pi}{4}} \frac{\sin \theta_1}{\cos^2 \theta_1} d\theta_1 \int_0^{\frac{\pi}{4}} \frac{\sin \theta_2}{\cos^2 \theta_2} d\theta_2 \\ &= \frac{1}{4} \left((\log(1 + \sqrt{2}))^2 + (\sqrt{2} - 1)^2 \right) \end{aligned}$$

残りの3つの範囲での積分も同様に、三角関数の有利関数の積分に帰着します。特に、対称性から $D_1 \times E_1$ 上の積分と $D_2 \times E_2$ 上での積分は同じ値になり、 $D_1 \times E_2$ 上の積分と $D_2 \times E_1$ 上での積分は同じ値になります。後者は以下のように求められます。

$$\iint \iint r_1 r_2 \cos(\theta_1 - \theta_2) dr_1 d\theta_1 dr_2 d\theta_2 = \frac{1}{\pi} (\sqrt{2} - 1) \log(1 + \sqrt{2})$$

4つの積分の値を合計することで、 $\cos \theta$ の期待値は

$$\int_0^1 \int_0^1 \int_0^1 \int_0^1 \frac{a_{11} a_{21} + a_{12} a_{22}}{\sqrt{(a_{11}^2 + a_{12}^2)(a_{21}^2 + a_{22}^2)}} da_{11} da_{21} da_{12} da_{22} = \frac{1}{2} (\log(1 + \sqrt{2}))^2 + \frac{1}{2} (\sqrt{2} - 1)^2 + (\sqrt{2} - 1) \log(1 + \sqrt{2})$$

と求められます。この値を計算すると

入力

```
mean_population_corr_coef = 1/2 * (np.log(1 + np.sqrt(2))) ** 2) +
                             1/2 * ((np.sqrt(2) - 1) ** 2) + (np.sqrt(2) - 1) * np.log(1 + np.sqrt(2))

print(mean_population_corr_coef)
```

出力

```
0.839273030836
```

となり、標本相関係数の100000個の平均値0.839にかなり近い値となりました。これで、標本相関係数が高い値をとる理由がわかりました。

さらに考察してみます。まず、「標本相関係数の100000個の平均値」と「 $\cos \theta$ の積分値」の差を確認してみましょう。

入力

```
# 小数第4位まで表示
%precision 4
corrcoef_ave[100000-1] - mean_population_corr_coef
```

出力

```
-0.0004
```

小さな差ですが、 n (今回は100000)をもっと大きくしていくことで、0に近づいていくのでしょうか？ 実は、 n をいくら大きくしても、「標本相関係数の平均値」は「 $\cos \theta$ の積分値」には収束しません。ここで、今回計算した標本相関係数 $\text{np.corrcoef}(X[:, 0], X[:, 1])[0, 1]$ とはどのようなものかを思い出してみましょう。今回の場合、標本相関係数 $\text{np.corrcoef}(X[:, 0], X[:, 1])[0, 1]$ は以下のように計算されます。

$$\frac{\frac{1}{200} \sum_{i=1}^{200} \left(x_{i1} - \frac{1}{200} \sum_{j=1}^{200} x_{j1} \right) \left(x_{i2} - \frac{1}{200} \sum_{k=1}^{200} x_{k2} \right)}{\sqrt{\frac{1}{200} \sum_{i=1}^{200} \left(x_{i1} - \frac{1}{200} \sum_{j=1}^{200} x_{j1} \right)^2} \sqrt{\frac{1}{200} \sum_{i=1}^{200} \left(x_{i2} - \frac{1}{200} \sum_{k=1}^{200} x_{k2} \right)^2}} = \frac{\frac{1}{200} \sum_{i=1}^{200} \left(a_1 b_i - \frac{1}{200} \sum_{j=1}^{200} a_1 b_j \right) \left(a_2 b_i - \frac{1}{200} \sum_{k=1}^{200} a_2 b_k \right)}{\sqrt{\frac{1}{200} \sum_{i=1}^{200} \left(a_1 b_i - \frac{1}{200} \sum_{j=1}^{200} a_1 b_j \right)^2} \sqrt{\frac{1}{200} \sum_{i=1}^{200} \left(a_2 b_i - \frac{1}{200} \sum_{k=1}^{200} a_2 b_k \right)^2}}$$

この標本相関係数が母相関係数 $\cos \theta$ のよい推定量となっているかが問題となります。2次元正規分布の場合、標本相関係数は母相関係数の最尤推定量にはなりますが、不偏推定量とはならず、以下の近似式が成り立つことが知られています^[*]。

$$E(r) \simeq \rho - \frac{\rho(1-\rho^2)}{2N}$$

(*) 参考文献 Hedges, Larry V.; Olkin, Ingram [1985]. Statistical Methods for Meta-Analysis. Academic Press. ISBN 0-12-336380-2. MR 0798597

ここで、 r は標本相関係数、 ρ は母相関係数、 N はサンプル数です。今回は $N = 200$ です。このように標本相関係数と母相関係数の間にはバイアスがあるため、 n をいくら大きくしてもバイアス分だけ差があります。このバイアス

$-\frac{\rho(1-\rho^2)}{2N}$ (近似値) を考慮すると、今回の標本相関係数の期待値は次のように修正されます。

$$\int_0^1 \int_0^1 \int_0^1 \int_0^1 \left(\cos \theta - \frac{\cos \theta}{2N} + \frac{\cos^3 \theta}{2N} \right) da_{11} da_{21} da_{12} da_{22}$$

ここで、 $\cos^3 \theta$ の積分が出てきましたが、この積分も $\cos \theta$ の積分と同様に $a_{11}, a_{12}, a_{21}, a_{22}$ を極座標変換することで、三角関数の有利関数の積分に帰します。その値は

$$\int_0^1 \int_0^1 \int_0^1 \int_0^1 \cos^3 \theta da_{11} da_{21} da_{12} da_{22} = \frac{21}{2} - 7\sqrt{2} + 3\log(1 + \sqrt{2}) - 3\sqrt{2}\log(1 + \sqrt{2}) + \frac{3}{2}(\log(1 + \sqrt{2}))^2$$

と計算できます。次に、実際にバイアスを計算してみましょう。

入力

```
cubed_mean = 21/2 - 7*np.sqrt(2) + 3*np.log(1 + np.sqrt(2)) -
              3*np.sqrt(2)*np.log(1 + np.sqrt(2))+3/2*(np.log(1 + np.sqrt(2)))**2
def bias(N):
    return -mean_population_corr_coef/(2*N) + cubed_mean/(2*N)
bias(200)
```

出力

-0.0004

上で求めた、「標本相関係数の100000個の平均値」と「 $\cos \theta$ の積分値」の差と同じ値となりました。「標本相関係数の100000個の平均値」、「 $\cos \theta$ の積分値」、「 $\cos \theta$ の積分値にバイアスを加えたもの」を比較してみましょう。

入力

```
print(' 標本相関係数の100000個の平均値:s=',corrcoef_ave[100000-1])
print('cos θ の積分値:t=',mean_population_corr_coef)
print('cos θ の積分値にバイアスを加えたもの:u=',mean_population_corr_coef+bias(200))
print('s-t=',corrcoef_ave[100000-1]-mean_population_corr_coef)
print('s-u=',corrcoef_ave[100000-1]-(mean_population_corr_coef+bias(200)))
```

出力

標本相関係数の100000個の平均値:s= 0.838868995091
cos θ の積分値:t= 0.839273030836
cos θ の積分値にバイアスを加えたもの:u= 0.838851106967
s-t= -0.000404035745149
s-u= 1.78881233176e-05

バイアスの影響を考慮した方が、標本相関係数の100000個の平均値に近いことが確かめられました。今回は N が200と大きいため、バイアスは小さな値となっていました。サンプル数を $N = 5$ にするとどうなるのでしょうか?その場合は

入力

bias(5)

出力

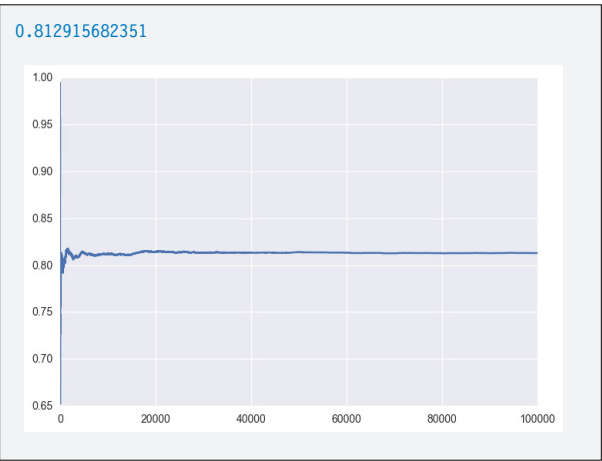
-0.0169

となり、 $N = 200$ の場合よりバイアスが大きくなります。さらに、標本相関係数の100000個の平均値を確かめてみます。

入力

```
corrcoef_sum = 0
corrcoef_ave = []
n =100000
N=5
for i in range(n):
    rng_data = np.random.RandomState(i)
    X = np.dot(rng_data.rand(2, 2), rng_data.randn(2, N)).T
    corrcoef_sum += np.corrcoef(X[:, 0], X[:, 1])[0,1]
    corrcoef_ave.append(corrcoef_sum / (i + 1 ))
plt.plot(corrcoef_ave)
print(corrcoef_ave[n-1])
```

出力



入力

```
print('100000 個の標本相関係数の平均値:s=',corrcoef_ave[100000-1])
print('cos  $\theta$  の積分値:t=',mean_population_corr_coef)
print('cos  $\theta$  の積分値にバイアスを加えたもの:u=',mean_population_corr_coef+bias(5))
print('s-t=',corrcoef_ave[100000-1]-mean_population_corr_coef)
print('s-u=',corrcoef_ave[100000-1]-(mean_population_corr_coef+bias(5)))
```

出力

```
100000 個の標本相関係数の平均値:s= 0.812915682351
cos  $\theta$  の積分値:t= 0.839273030836
cos  $\theta$  の積分値にバイアスを加えたもの:u= 0.822396076097
s-t= -0.0263573484848
s-u= -0.00948039374613
```

$N = 5$ でも、バイアスの影響を考慮した方が、100000 個の平均値に近いことが確かめられました。

※なお、本補足説明は石橋佳久さんの協力によって作成されました。ありがとうございました。