

赵卫东

计算机科学技术学院副教授

硕士生导师

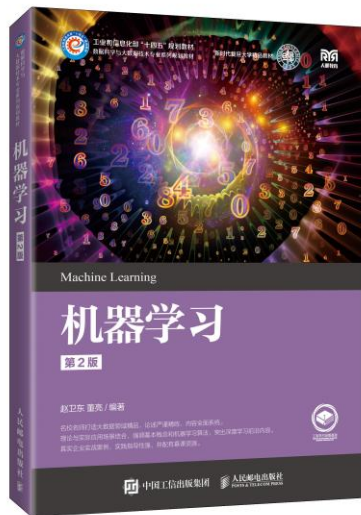
复旦大学

oneAPI 驱动机器学习加速

- 机器学习课程在复旦大学开设，复旦大学是中国顶尖的本科生和研究生大学之一。该课程涵盖机器学习和深度学习的基础知识；
- 将分享本课程中基于案例的学习经验：利用循序渐进的方法来分析问题并应用机器/深度学习理论和算法来解决挑战；
- 展示在解决问题的过程中在特定案例中使用英特尔 oneAPI 人工智能分析工具套件组件的体验，当正确的硬件和软件组件协同工作以提高在客户端或服务器上运行的代码的工作效率时，学生将享受到性能提升；
- 介绍学生如何积极参与英特尔黑客马拉松活动，将课堂上学到的技能运用到实际中，并利用机器学习、数据科学和人工智能应对现实生活中的挑战



赵教授于2003年加入复旦大学，在复旦大学计算机学院工作。他在大学为本科生和研究生提供机器学习、深度学习、大数据分析和商业智能课程。在数据科学、机器和深度学习的多个领域发表了 100 多篇论文和 10 多本书籍。



使用oneAPI实现机器学习加速

赵卫东 复旦大学

- Numpy数据处理加速
- Pandas数据处理加速
- Scikit-learn算法加速
- 实践平台操作介绍

模型加速和数据处理加速

- 模型加速要求
 - 加速效果要明显（按数量级性能提升）
 - 对原有代码的改动要少（代码改动少、结构不变）
 - 对未来的适应性好(可移植到新的硬件)
- 目标：
 - 分析 Numpy、Pandas等常见数据处理代码，找到性能低下的循环和列表处理，并用速度更快的矢量化进行改进；
 - 调整Scikit-learn算法，优化其在CPU、GPU 等加速器设备的表现

Numpy数据处理加速

目标：应用NumPy广播、聚合、通用函数、where和select子句加速Python循环

学习内容：

- 低效代码有哪些：如耗时循环，浪费资源，浪费时间
- Numpy 加速代码的方式
- 在各种场景中应用循环替换方法

Numpy数据处理加速-替换循环

实现一个循环处理任务：

1. 创建一个包含 100 万个随机浮点数的列表。
2. 将使用 for 循环遍历其元素，获取 Log10 并将值存储在另一个列表中。
3. 计算原生For循环、map函数、list推导、numpy操作耗时

准备工作：

引入相关库：

```
import numpy as np
from math import log10 as lg10
import time
import matplotlib.pyplot as plt
import random
%matplotlib inline
```

生成100万个随机浮点数：

```
N = 100000000 # Number of records to process
acceleration = [] # Empty list to store operation speeds (time taken)
l1 = list(100*(np.random.random(N))+1)
a1 = np.array(l1)
```

赵卫东 复旦大学

替换For循环-结果对比

Python原生For循环方法

```
1 timing = {} # 记录不同方法耗时
2
3 t1=time.time()
4 l2 = []
5 for item in l1:
6     l2.append(lg10(item))
7 t2 = time.time()
8 print("原生Loop耗时 {} 秒".format(t2-t1))
9 timing['loop'] = (t2-t1)
10 print("前4条数据:", l2[:4])
```

这种方法耗时最长

原生Loop耗时 13.897366046905518 秒

前4条数据: [1.5835638721530192, 1.681212234146597, 1.9828731712633745, 1.3225891470348725]

使用Map 函数

```
1 def op1(x):
2     return (lg10(x))
3
4 t1=time.time()
5
6 l2=list(map(op1,l1))
7
8 t2 = time.time()
9 print("list(map)耗时{} 秒".format(t2-t1))
10 timing['map'] = (t2-t1)
11 print("前4条数据:", l2[:4])
```

list(map)耗时12.301867961883545 秒

前4条数据: [1.5835638721530192, 1.681212234146597, 1.9828731712633745, 1.3225891470348725]

替换For循环-结果对比

使用 *List* 推导方法

```
1 t1=time.time()
2 l2 = [lg10(l1[i]) for i in range(len(l1))]
3 t2 = time.time()
4 print("list comprehension耗时{} seconds".format(t2-t1))
5 timing['list comprehension'] = (t2-t1)
6 print("前4条数据:", l2[:4])
```

list comprehension耗时12.51164984703064 seconds

前4条数据: [1.5835638721530192, 1.681212234146597, 1.9828731712633745, 1.3225891470348725]

使用 *NumPy* 向量化方法

```
1 t1=time.time()
2 a2=np.log10(a1)
3 t2 = time.time()
4 print("NumPy log10方法耗时 {} seconds".format(t2-t1))
5 timing['numpy'] = (t2-t1)
6 l3 = list(a2)
7 print("前4条数据:", l3[:4])
```

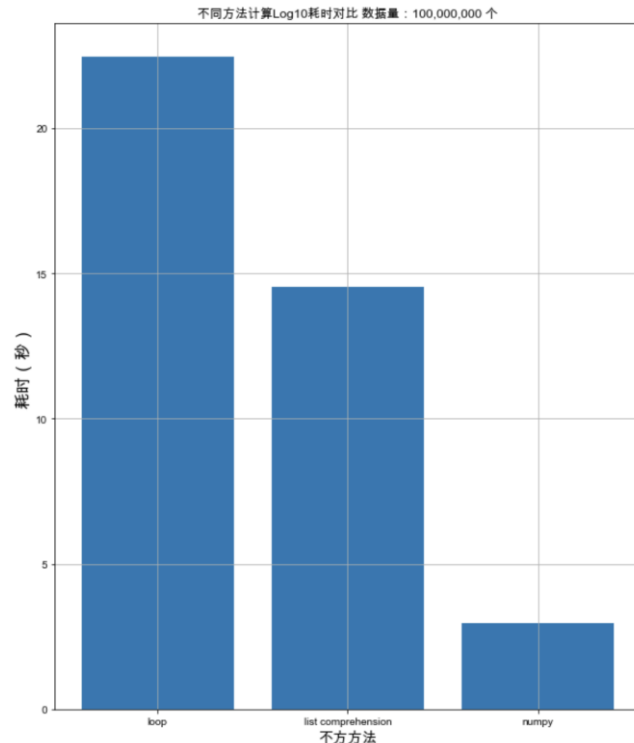
NumPy log10方法耗时 1.118406057357788 seconds

前4条数据: [1.5835638721530192, 1.681212234146597, 1.9828731712633745, 1.3225891470348725]

不同方法时间对比

```
1 %matplotlib inline
2 import matplotlib.pyplot as plt
3 plt.figure(figsize=(16,6))
4 plt.title("不同方法计算Log10耗时对比 数据量: {}, 个".format(a1.shape[0]))
5 plt.ylabel("耗时 (秒)", fontsize=12)
6 plt.xlabel("方法", fontsize=14)
7 plt.grid(True)
8 plt.bar(x = range(len(timing)), height=timing.values(), align='center', tick_label=list(timing.keys()))
9 print('速度提升: {:.40f} 倍'.format(timing['loop']/timing['numpy']))
```

速度提升: 12 倍



NumPy对ndarray对象的操作比Python数学操作快得多。因为矢量化

操作可在内存布局和迭代中数据类型解释等方面进行优化。

平移列表中的元素

在时间序列预测中，我们通常希望在给定较早的可用特征数据的情况下预测时间序列中较晚的值。
需要选择一个时移的目标变量来及时预测后面的标签。

目标：将一系列数值中所有数移动一个常数位置

准备数据

```
1 num = 100000001
2 a = np.linspace(0, num - 1, num=num)
3 b = np.ndarray(num-1)
4
5 timing = {}
```

Python原生For循环方法

```
1 t1=time.time()
2
3 for i in range(len(a)-1):
4     b[i] = a[i+1]
5
6 t2=time.time()
7 print("原生方法耗时 {} 秒".format(t2-t1))
8 timing['loop'] = (t2-t1)
9 b[:2], b[-2:]
```

原生方法耗时 22.482749938964844 secs

(array([1., 2.]), array([9.9999999e+07, 1.0000000e+08]))

使用 *List* 推导方法

```
1 t1=time.time()
2
3 b = [a[i+1] for i in range(len(a)-1)] # shift b by 1
4
5 t2=time.time()
6 print("耗时 {} 秒".format(t2-t1))
7 timing['list comprehension'] = (t2-t1)
8 b[:2], b[-2:]
```

shift b 14.524049043655396 secs

([1.0, 2.0], [99999999.0, 100000000.0])

平移列表中的元素

使用Numpy ufuncs 方法

```
1 t1=time.time()
2
3 b = np.delete( np.roll(a,-1) , -1) # numpy roll and delete last value
4
5 t2=time.time()
6 print("耗时 {} 秒".format(t2-t1))
7 timing['numpy'] = (t2-t1)
8 b[:2], b[-2:]
```

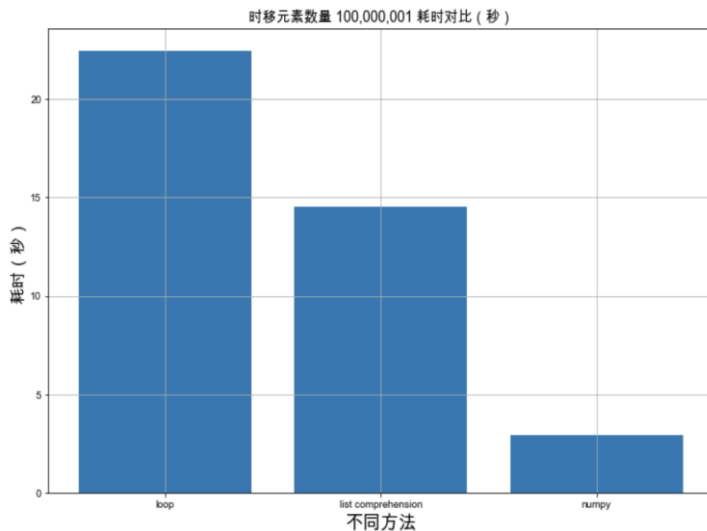
耗时 2.94555401802063 秒

(array([1., 2.]), array([9.9999999e+07, 1.0000000e+08]))

什么是ufunc?

通用函数（或简称为 ufunc）是一种以逐个元素的方式对 ndarrays 进行操作的函数，支持数组广播、类型转换和其他一些标准功能。采用固定数量的输入和固定数量的输出。

结果：numpy性能提升7倍



NumPy中的通用函数 (ufunc)

目前NumPy 中定义了 60 多个通用函数，例如：

- **Methods for reductions:** reduce, accumulate, outer, ...
- **Math Functions:** log, exp, matmul, mod, multiply, add, ...
- **Trigonometric functions:** sin, cosine, tan, arcsin, hypot, ...
- **Bit-twiddling functions:** bitwise_and, bitwise_or, invert, left_shift, ...
- **Comparison functions:** greater, less_than, not_equal, logical_and ...
- **Floating functions:** isfinite, isinf, isnan, isnat, fabs, ...

Functions	Description
np.mean()	Compute the arithmetic mean along the specified axis.
np.std()	Compute the standard deviation along the specified axis.
np.var()	Compute the variance along the specified axis.
np.sum()	Sum of array elements over a given axis.
np.prod()	Return the product of array elements over a given axis.
np.cumsum()	Return the cumulative sum of the elements along a given axis.
np.cumprod()	Return the cumulative product of elements along a given axis.
np.min(), np.max()	Return the minimum / maximum of an array or minimum along an axis.
np.argmin(), np.argmax()	Returns the indices of the minimum / maximum values along an axis
np.all()	Test whether all array elements along a given axis evaluate to True.
np.any()	Test whether any array element along a given axis evaluates to True.

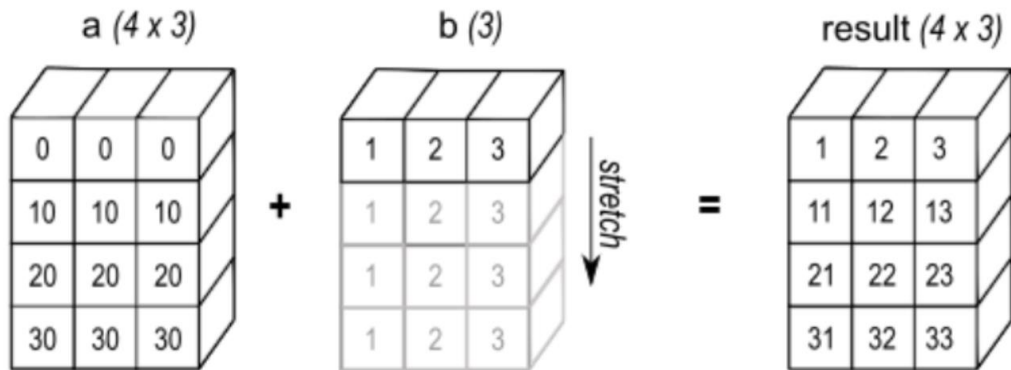
许多 ufunc 内置了可以应用的归约(reductions)：

- Python 函数 max() 将在一维数组中找到最大值，但使用较慢的序列接口来实现。
- 采用ufunc 的 reduce 方法要快得多。
- 对于大于1维的数组，max() 方法可能会给出错误答案。
- minimum 的 reduce 方法还可用于计算数组的总最小值。

NumPy中的广播 (Broadcasting)

Numpy广播:

在算术运算期间如何处理具有不同形状的数组。在某些约束条件下，较小的数组在较大的数组中“广播”，以便它们具有兼容的形状。



通过从较小维度复制数据直到生成的数组适合线性代数加法或乘法，从而使维度成为正确大小的方法，从而加快计算速度。

NumPy中的广播 (Broadcasting)

播出规则及条件:

1. 数组的形状相同或两个数组的结束维度匹配

```
1 # 维度相同
2 x = np.array([3,2,1])
3 y = np.array([6,5,2])
4 z = x * y
5 print("x.shape is", x.shape)
6 print("y.shape is", y.shape)
7 print("x.shape == y.shape:", x.shape == y.shape)
8 print("z: ", z)
```

```
x.shape is (3,)
y.shape is (3,)
x.shape == y.shape: True
z: [18 10 2]
```

2. 如果一个数组的维数与另一个数组的维数不同, 则将维数较小的数组扩展 1, 直到两个数组的维数相等

```
1 x = np.array([3,2,1,2,6,4]*3).reshape(6,3)
2 print("x: \n", x)
3 y = np.array([1,2,3]).reshape(1,3)
4 print("y: \n", y)
5 print("len(x)", len(x))
6 print("len(y)", len(y))
7 print("x - y\n", x - y)
8
```

```
x:
[[3 2 1]
 [2 6 4]
 [3 2 1]
 [2 6 4]
 [3 2 1]
 [2 6 4]]
```

```
y:
[[1 2 3]]
len(x) 6
len(y) 1
x - y
[[ 2  0 -2]
 [ 1  4  1]
 [ 2  0 -2]
 [ 1  4  1]
 [ 2  0 -2]
 [ 1  4  1]]
```

3. 如果要广播两个数组, 则维度大小为 1 的数组的行为方式应使维度较小的数组能够通过复制扩展到第二个数组的长度。

```
1 # Condition 2
2 x = np.array([3,2,1,2,6,4,5,7,9]).reshape(3,3)
3 y = np.array([1,2,3])
4 print("x: \n", x)
5 print("y: \n", y)
6 print("y.shape extended by one internally (3,1)\n")
7
8 z = x * y
9 print("z: \n", z)
10 # 按行相乘:
11 # [3,2,1] x [1,2,3] = [3,4,3]
```

```
x:
[[3 2 1]
 [2 6 4]
 [5 7 9]]
y:
[1 2 3]
y.shape extended by one internally (3,1)
```

```
z:
[[ 3  4  3]
 [ 2 12 12]
 [ 5 14 27]]
```

NumPy中的广播性能比较

规范化加州住房数据集的示例（结合了 Numpy 聚合和广播）

2种方法定义：

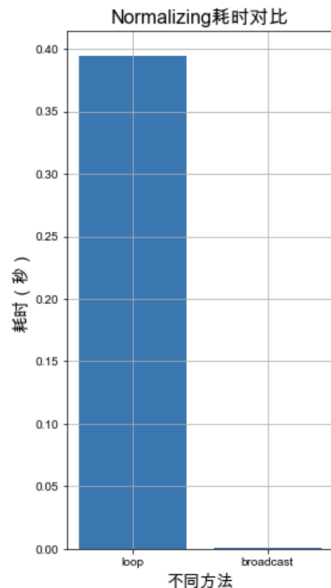
```
1 from sklearn.datasets import fetch_california_housing
2
3 california_housing = fetch_california_housing(as_frame=True)
4 X = california_housing.data.to_numpy()
5 print("data shape:", X.shape)
6 def standardizeLoop(B):# 方法一：原生循环
7     L = len(B)
8     tmp = np.zeros(B.shape)
9     for col in range(X.shape[1]):
10         mean = 0
11         total = 0
12         for row in range(L): # compute mean for col
13             total += B[row, col]
14         mean = total/L
15         diff2 = 0
16         for row in range(L):
17             d = B[row,col] - mean
18             diff2 += d*d
19         tmp[row, col] = B[row, col] - mean
20     for row in range(L):
21         tmp[row, col] = tmp[row, col]/np.sqrt(diff2/(L))
22     return tmp
23
24 def standardize(A): # 方法二：numpy广播
25     return (A - A.mean(axis=0))/A.std(axis=0)
```

计算耗时和可视化：

```
1 t1 = time.time()
2 standardizeLoop(X)
3 t2 = time.time()
4 loop = t2-t1
5 timing['loop'] = loop
6 print("Python原生方法, 耗时 (秒) : ", loop)
7 t1 = time.time()
8 standardize(X)
9 t2 = time.time()
10 loop = t2-t1
11 timing['broadcast'] = loop
12 print("Numpy中broadcast方法, 耗时 (秒) : ", loop)
13
14
15 %matplotlib inline
16 import matplotlib.pyplot as plt
17 plt.figure(figsize=(12,8))
18 plt.title("Normalizing耗时对比", fontsize=16)
19 plt.ylabel("耗时 (秒)", fontsize=14)
20 plt.xlabel("不同方法", fontsize=14)
21 plt.grid(True)
22 plt.bar(x = list(timing.keys()), height=list(timing.values()),
23         align='center', tick_label=list(timing.keys()))
24 print("加速 {:.40f} 倍".format( timing['loop']/timing['broadcast']))
```

Python原生方法, 耗时 (秒) : 0.4358561038970947
Numpy中broadcast方法, 耗时 (秒) : 0.000982046127319336
加速 444 倍

实验结果：



通过对比可以发现：Numpy 向量化代码更具可读性，更易于调试和解释。

NumPy中的矩阵相乘

在机器学习里面经常涉及到矩阵相乘，例如：

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

任务： 构造2个500X500的浮点数矩阵，并计算它们的矩阵相乘所耗费的时间。

构造2个500X500的浮点数矩阵

准备数据：

```
1 import numpy as np
2 import time
3
4 N = 500
5 a = np.random.rand(N, N)
6 b = np.random.rand(N, N)
7 speed = {}
```

NumPy中的矩阵相乘

Naive triply nested loop

```
1 t1=time.time()
2 N = a.shape[0]
3 c = np.zeros_like(a)
4 for i in range(N):
5     for k in range(N):
6         for j in range(N):
7             c[i,j] += a[i,k] * b[k,j]
8 t2=time.time()
9 print("Matrix Mult Triply Nested Loop: {} secs".format(t2-t1))
10 speed['Triply Nested'] = t2-t1
11 c
```

Matrix Mult Triply Nested Loop: 74.34173488616943 secs

```
array([[123.38649331, 126.25889035, 126.08595224, ..., 123.38936178,
        121.21418327, 121.99053732],
       [123.82188641, 134.22201076, 131.38022436, ..., 121.97395313,
        124.93784992, 124.2379165 ],
       [132.27753185, 131.33585446, 132.17853368, ..., 125.25122153,
        122.04137968, 124.32192571],
       ...,
       [129.32599692, 132.28420158, 129.31854742, ..., 126.28999651,
        122.19144764, 128.29237107],
       [125.5719987 , 127.35888273, 125.88333694, ..., 121.19943408,
        118.10673515, 122.31089254],
       [123.44531583, 130.60344488, 126.21137012, ..., 123.90428512,
        123.1165294 , 122.46607358]])
```

这种方法最笨重，也最慢

Numpy.dot method to multiply arrays

```
1 # np.multiply explicitly with numpy ufunc
2 t1=time.time()
3
4 c = np.dot(a,b)
5
6 t2=time.time()
7 print("With np.dot(a.T,b): {} secs".format(t2-t1))
8 speed['np.dot(a.T,b)'] = t2-t1
9 c
```

With np.dot(a.T,b): 0.00731205940246582 secs

```
array([[123.38649331, 126.25889035, 126.08595224, ..., 123.38936178,
        121.21418327, 121.99053732],
       [123.82188641, 134.22201076, 131.38022436, ..., 121.97395313,
        124.93784992, 124.2379165 ],
       [132.27753185, 131.33585446, 132.17853368, ..., 125.25122153,
        122.04137968, 124.32192571],
       ...,
       [129.32599692, 132.28420158, 129.31854742, ..., 126.28999651,
        122.19144764, 128.29237107],
       [125.5719987 , 127.35888273, 125.88333694, ..., 121.19943408,
        118.10673515, 122.31089254],
       [123.44531583, 130.60344488, 126.21137012, ..., 123.90428512,
        123.1165294 , 122.46607358]])
```

NumPy中的矩阵相乘

使用Numpy.matmul to multiply arrays

```
1 # np.multiply explicitly with numpy ufunc
2 t1=time.time()
3
4 c = np.matmul(a,b)
5
6 t2=time.time()
7 print("With np.matmul(a,b): {} secs".format(t2-t1))
8 speed['np.matmul(a,b)'] = t2-t1
9 c
```

With np.matmul(a,b): 0.00806879997253418 secs

```
array([[123.38649331, 126.25889035, 126.08595224, ..., 123.38936178,
        121.21418327, 121.99053732],
       [123.82188641, 134.22201076, 131.38022436, ..., 121.97395313,
        124.93784992, 124.2379165 ],
       [132.27753185, 131.33585446, 132.17853368, ..., 125.25122153,
        122.04137968, 124.32192571],
       ...,
       [129.32599692, 132.28420158, 129.31854742, ..., 126.28999651,
        122.19144764, 128.29237107],
       [125.5719987 , 127.35888273, 125.88333694, ..., 121.19943408,
        118.10673515, 122.31089254],
       [123.44531583, 130.60344488, 126.21137012, ..., 123.90428512,
        123.1165294 , 122.46607358]])
```

这种方法与np.dot效果差不多

使用 Scipy的Matrix Multiply

```
1 import scipy.linalg.blas as LA
2 t1 = time.time()
3 B = LA.sgemm(alpha=1.0, a=a, b=b)
4 t2 = time.time()
5 print(t2-t1, "[s]")
6 speed['Scipy LA'] = t2-t1
7 B
```

0.004348039627075195 [s]

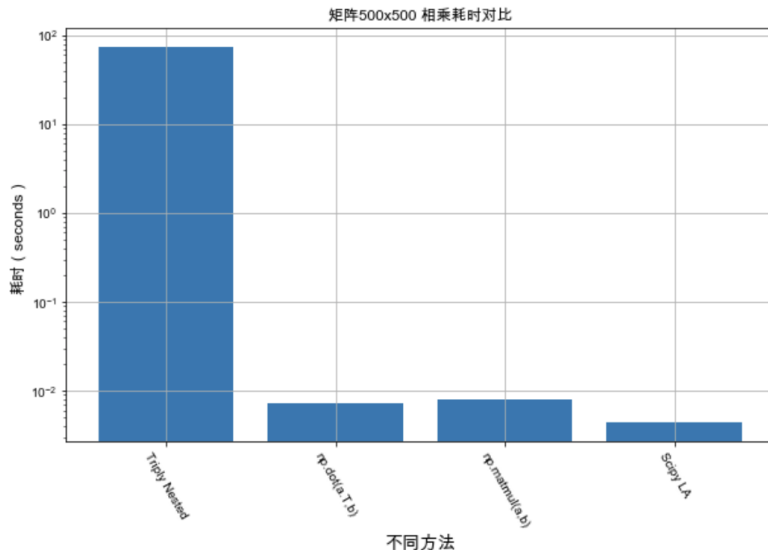
```
array([[123.38649 , 126.258896, 126.08594 , ..., 123.38936 , 121.21418 ,
        121.99055 ],
       [123.821884, 134.22202 , 131.38025 , ..., 121.97395 , 124.937836,
        124.237946],
       [132.27756 , 131.33586 , 132.17853 , ..., 125.2512 , 122.0414 ,
        124.32191 ],
       ...,
       [129.32597 , 132.28421 , 129.31854 , ..., 126.289986, 122.19144 ,
        128.29234 ],
       [125.57196 , 127.35889 , 125.88333 , ..., 121.19945 , 118.10673 ,
        122.31088 ],
       [123.44531 , 130.60347 , 126.21138 , ..., 123.90427 , 123.11655 ,
        122.466064]], dtype=float32)
```

这是Scipy库中的方法，相对也是很快速。

NumPy中的矩阵相乘

```
1 import matplotlib.pyplot as plt
2
3 timing = list(speed.values())
4 algo = list(speed.keys())
5
6 plt.figure(figsize=(10,6))
7 plt.title("矩阵{}x{} 相乘耗时对比".format(N,N), fontsize=12)
8 plt.ylabel("耗时 (seconds)", fontsize=12)
9 plt.xlabel("不同方法", fontsize=14)
10 plt.grid(True)
11 plt.yscale('log')
12 plt.xticks(rotation=-60)
13 bars = [i+1 for i in range(len(speed))]
14 timing = [v for v in speed.values()]
15 print("耗时:", timing)
16 algos = [v for v in speed.keys()]
17 print(algos)
18 plt.bar(x = bars, height= timing, align='center', tick_label= algos)
19 speedup = timing[0]/min(timing)
20
21 print("加速 {:,} 倍".format(int(np.round(speedup))))
22
```

耗时: [74.34173488616943, 0.00731205940246582, 0.00806879997253418, 0.004348039627075195]
['Triply Nested', 'np.dot(a.T,b)', 'np.matmul(a,b)', 'Scipy LA']
加速 17,098 倍



实验结果可视化

- 尽量使用已经分析和优化过的循环替换原生循环。
- 矩阵乘法已经很成熟，可选任意一个快速版本使用。

Pandas数据处理加速

学习目标:

- 应用 Numpy 方法来显著加速某些常见的 Pandas 瓶颈
- 在由 oneAPI 提供支持的 Numpy 中应用 WHERE 或 SELECT
- 使用 Numpy 技术避免 iterrows
- 通过将数字列转换为 numpy 数组来获得更好的性能

思路:

- 将 NumPy Select 和 Where 子句扩展到 Pandas 数据框 “应用” 函数
- 在应用函数调用的复杂条件逻辑上实现相当大的加速。

Pandas数据处理加速

目标：研究将自定义函数应用于数据集所有行的不同时间。

方法：循环的每次迭代中为所有数据元素执行相同的指令操作（取log值）。

构造一个100000X5的DataFrame

```
1 import pandas as pd
2 import numpy as np
3 import time
4
5 BIG = 100_000
6 df = pd.DataFrame(np.random.randint(0, 11, size=(BIG, 5)),
7                   columns=('a', 'b', 'c', 'd', 'e'))
8 df.head()
```

定义一个取log的操作

```
1 def my_function(x):
2     return np.log(1+x)
```

生成模拟数据

	a	b	c	d	e
0	9	0	6	2	7
1	10	2	8	3	4
2	3	6	10	4	10
3	10	9	3	3	0
4	6	3	5	0	0

Pandas数据处理加速

使用iloc方法

```
1 %%time
2 # naive loop method using pandas iloc
3 timing = {}
4 t1 = time.time()
5
6 for i in range(0,BIG):
7     df.iloc[i,2] = my_function(df.iloc[i,0])
8
9 t2 = time.time()
10 timing['iloc'] = t2 - t1
11 df.head()
```

CPU times: user 25.1 s, sys: 45.8 ms, total: 25.1 s
Wall time: 25.2 s

	a	b	c	d	e
0	9	2	2.302585	0	7
1	8	2	2.197225	8	4
2	5	3	1.791759	1	5
3	8	1	2.197225	2	10
4	5	10	1.791759	8	2

使用Pnadas iterrow方法

```
1 %%time
2 # naive loop method using pandas loc
3 import numpy as np
4 import time
5
6 # each iteration of the loop requires
7 # an interpretation of the instructions
8 # being used and this decoding takes time
9 t1 = time.time()
10
11 for index, row in df.iterrows():
12     row[2] = my_function(row[0])
13
14 t2 = time.time()
15 baseTime = t2-t1
16 timing['iterrow'] = t2 - t1
17 df.head()
```

CPU times: user 5.25 s, sys: 9.72 ms, total: 5.25 s
Wall time: 5.26 s

	a	b	c	d	e
0	9	2	2.302585	0	7
1	8	2	2.197225	8	4
2	5	3	1.791759	1	5
3	8	1	2.197225	2	10
4	5	10	1.791759	8	2

iloc方法更快

Pandas数据处理加速

使用Pandas"at"方法

```
1 %%time
2 # naive loop method using pandas "at"
3 t1 = time.time()
4
5 for i in range(0,BIG):
6     df.at[i,'c']=my_function(df.at[i,'a'])
7
8 t2 = time.time()
9 fastest_time = t2-t1
10 Speedup = baseTime / fastest_time
11 print("Speed up: {:.40f} X".format(Speedup))
12 timing['df.at'] = t2 - t1
13 df.head()
```

Speed up: 4 X
CPU times: user 1.26 s, sys: 5.12 ms, total: 1.27 s
Wall time: 1.27 s

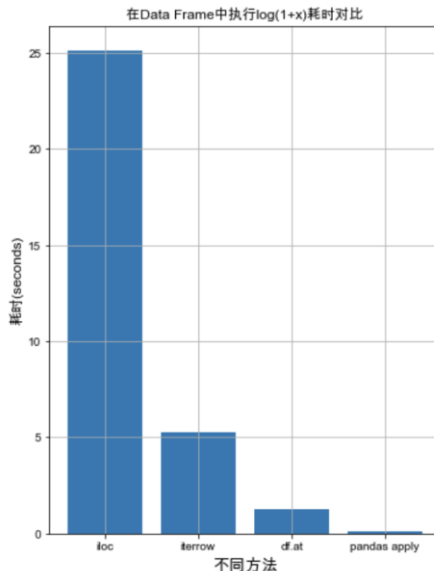
	a	b	c	d	e
0	9	2	2.302585	0	7
1	8	2	2.197225	8	4
2	5	3	1.791759	1	5
3	8	1	2.197225	2	10
4	5	10	1.791759	8	2

使用Pandas的Apply方法

```
1 %%time
2 # vectorized method using pandas apply
3 t1 = time.time()
4
5 df['c']=df['a'].apply(lambda x : my_function(x))
6
7 t2 = time.time()
8 fastest_time = t2-t1
9 Speedup = baseTime / fastest_time
10 timing['pandas apply'] = t2 - t1
11
12 print("Speed up: {:.40f} X".format(Speedup))
```

Speed up: 40 X
CPU times: user 129 ms, sys: 3.54 ms, total: 132 ms
Wall time: 131 ms

Pandas的apply方法性能最优!



```
1 %matplotlib inline
2 import matplotlib.pyplot as plt
3 plt.figure(figsize=(10,6))
4 plt.title("在Data Frame中执行log(1+x)耗时对比")
5 plt.ylabel("耗时(seconds)", fontsize=12)
6 plt.xlabel("不同方法", fontsize=14)
7 plt.grid(True)
8 plt.bar(x = range(len(timing)), height=timing.values(), align='center', tick_label=list(timing.keys()))
9 short = min(list(timing.values()))
10 long = max(list(timing.values()))
11 print("加速 {:.40f} 倍".format(long/short))
```

加速 : 192 倍

Pandas数据处理加速

问题：当样本量再增大时，元素遍历性能会是什么样的？

生成数据集：

构建一个400万条样本的DataFrame

```
1 import pandas as pd
2 import numpy as np
3 timing = {}
4 BIG = 4000000
5 df = pd.DataFrame(np.random.randint(0, 11, size=(BIG, 5)),
6                   columns=('a', 'b', 'c', 'd', 'e'))
7 print("data type:", type(df['a'].values))
8 df.head()
```

data type: <class 'numpy.ndarray'>

	a	b	c	d	e
0	2	10	1	2	10
1	4	9	10	8	3
2	4	5	0	4	9
3	8	6	10	4	8
4	8	5	1	0	1

定义对DataFrame 执行条件逻辑操作

```
1 def func(a,b,c,d,e):
2     if e == 10:
3         return c*d
4     elif (e < 10) and (e>=5):
5         return c+d
6     elif e < 5:
7         return a+b
```

Pandas数据处理加速

采用Pandas的Apply方法

```
1 import time
2 t1 = time.time()
3 df['new'] = df.apply(lambda x: func(x['a'], x['b'], x['c'], x['d'], x['e']), axis=1)
4 t2 = time.time()
5 print("time : {:.5f}".format(t2-t1))
6 df.head()
7 timing['Pandas Apply'] = t2 - t1
8 baseTime = t2-t1
```

time : 65.21

使用向量化方法

```
1 t1 = time.time()
2 df['new'] = df['c'] * df['d'] #default case e = 10
3 mask = (df['e'] < 10) & (df['e'] >= 5)
4 df.loc[mask, 'new'] = df['c'] + df['d']
5 mask = df['e'] < 5
6 df.loc[mask, 'new'] = df['a'] + df['b']
7 t2 = time.time()
8 print("time :", t2-t1)
9 fastest_time = t2-t1
10 timing['Mask'] = t2 - t1
11 Speedup = baseTime / fastest_time
12 print("Speed up: {:.4f} X".format(Speedup))
13 df.head()
```

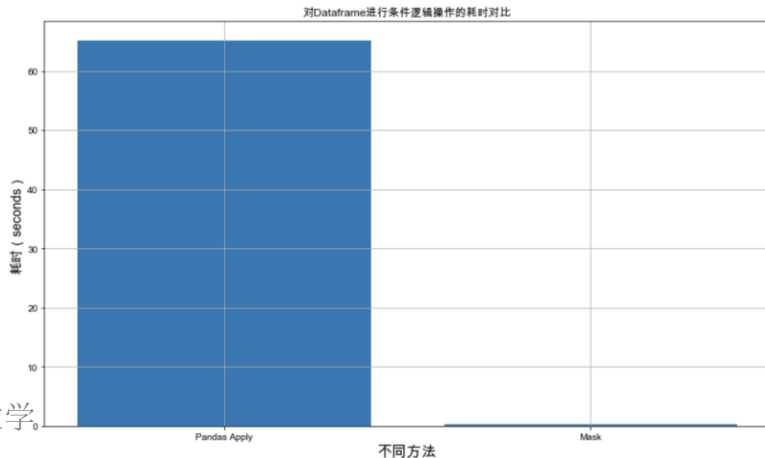
time : 0.3452107906341553
Speed up: 189 X

	a	b	c	d	e	new
0	2	10	1	2	10	2
1	4	9	10	8	3	13
2	4	5	0	4	9	4
3	8	6	10	4	8	14
4	8	5	1	0	1	13

结果可视化:

```
1 %matplotlib inline
2 import matplotlib.pyplot as plt
3 plt.figure(figsize=(10,8))
4 plt.title("对DataFrame进行条件逻辑操作的耗时对比")
5 plt.ylabel("耗时 (seconds)", fontsize=14)
6 plt.xlabel("不同方法", fontsize=16)
7 plt.grid(True)
8 plt.bar(x = range(len(timing)), height=timing.values(),
9         align='center', tick_label=list(timing.keys()))
10 short = min(list(timing.values()))
11 long = max(list(timing.values()))
12 print('加速 : {:.4f} 倍'.format(long/short))
```

加速 : 189 倍



采用向量化, 速度更快!

赵卫东 复旦大学

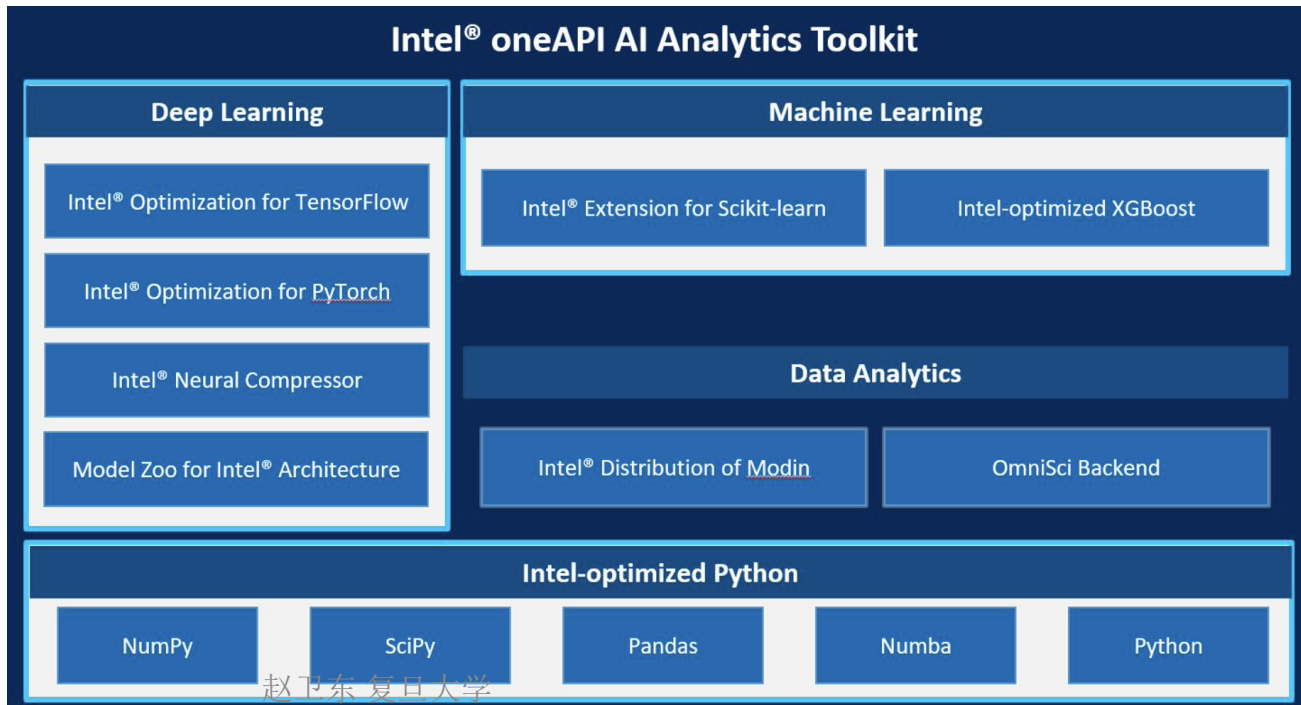
Scikit-learn算法加速

英特尔® oneAPI 人工智能分析工具包

- 使用 oneAPI 库构建，用于低级计算优化。
- 最大限度地提高从预处理到机器学习的性能。

特点：

- 快速推理集成：
TensorFlow* 和 PyTorch*
、预训练模型
- Modin*、scikit-learn* 和
针对英特尔优化的
XGBoost*



Scikit-learn算法加速-环境安装

推荐环境安装方法： `pip install scikit-learn-intelex`

- 方法二：
- Install from **Anaconda Cloud: Conda-Forge channel** (recommended by default for conda users):
 - into a newly created environment (recommended):

```
conda create -n env -c conda-forge python=3.9 scikit-learn-intelex
```

! Important

If you do not specify the version of Python (`python=3.9` in the example above), then Python 3.10 is downloaded by default, which is not supported.
See [supported configurations for Conda-Forge channel](#).

- into your current environment:

```
conda install scikit-learn-intelex -c conda-forge
```

Scikit-learn算法加速-环境安装

方法三：

Install from **Anaconda Cloud: Main channel:**

- into a newly created environment (recommended):

```
conda create -n env python=3.9 scikit-learn-intelex
```

! Important

If you do not specify the version of Python (**python=3.9** in the example above), then Python 3.10 is downloaded by default, which is not supported.

See [supported configurations for Anaconda main channel](#).

- into your current environment:

```
conda install scikit-learn-intelex
```

方法四：

```
conda create -n env -c conda-forge python=3.9 scikit-learn-intelex
```

```
conda install scikit-learn-intelex -c conda-forge
```

还可以源码编译：参考<https://intel.github.io/scikit-learn-intelex/installation.html>

Scikit-learn算法加速-已优化算法

Scikit-learn库中已优化的算法列表（一）

- 针对 Intel CPU 和 Intel GPU 优化了不同的算法
- 可查看优化了哪些功能
- 有些函数只是该列表中其他函数的别名

	Acronym	Name	What is it used for	Advantages	Disadvantages
0	config_context	config_context	We will ignore this one for the course, but it...	NaN	NaN
1	dbscan	Density-based spatial clustering of applicatio...	Density-based spatial clustering of applicatio...	does not require one to specify the number of ...	sklearnex must only use 'euclidean' or 'minko...
2	distances	Distances	AKA Pairwise Distnace: Part of the sklearn.met...	Used to find similarity of vectors - such as s...	Intel Extensions for scikit-learn:\nWith metri...
3	elasticnet	ElasticNet	Linear regression with combined L1 and L2 prio...	The elastic net method performs variable selec...	Intel sklearnex: All parameters except sample_...
4	fin_check	aka assert_all_finite	Throw a ValueError if X contains NaN or infinity.	Can be very valuable when doing data cleaning ...	Only dense data is supported
5	get_config	get_config	Retrieve current values for configuration set ...	NaN	NaN
6	kmeans	K-Means clustering.	KMeans algorithm clusters data by trying to se...	Relatively simple to implement. Scales to larg...	intel sklearnex: All parameters except precomp...
7	kneighborsclassifier	K Nearest Neighbors Classifier	Classifier: Find the K-neighbors of a point. R...	K-NN is pretty intuitive and simple: \nK-NN al...	Intel sklearnx: 'euclidean' or 'minkowski' wi...
8	kneighborsregressor	K Nearest Neighbors Regressor	Regressor: Find the K-neighbors of a point. Re...	K-NN is pretty intuitive and simple: \nK-NN al...	Intel sklearnx: 'euclidean' or 'minkowski' wi...
9	knn_classifier	k nearest neighbors classifier aka KNN classifier	Classifier implementing the k-nearest neighbor...	Advantages of KNN\nNo Training Period: KNN is ...	optimal choice of the value is highly data-dep...
10	knn_regressor	k nearest neighbors regressor aka KNN regressor	Same as classifier	same as classifier	same as classifier
11	lasso	Lasso	for Regression: The Lasso is a linear model th...	Select features, by shrinking co-efficient tow...	Intel sklearnex: All parameters except sample_...
12	linear	Linear Regression	Fitting line of best fit through data	Simple method \nGood interpretation \nEasy to...	Intel sklearnex: All parameters except normali...
13	log_reg	alias for Logistic Regression	same as below	same as below	same as below
14	logistic	Logistic Regression	Classifier: Generally used for quick binary cl...	Doesn't assume linear relationship between ind...	Intel sklearnex: suggest solver: 'lbfgs' or 'n...
15	logisticregression	alias for Logistic Regression	same as above	same as above	same as above

Scikit-learn算法加速-已优化算法

Scikit-learn库中已优化的算法列表（二）

16	nearest_neighbors	Nearest Neighbor	Unsupervised learner: Unsupervised nearest ne...	it is unsupervised - requires no y: \nFor big ...	sklearnex must only use 'euclidean' or 'minko...
17	nearestneighbors	alias for Nearest Neighbor	same as above	same as above	same as above
18	nusvc	nuSVC	Similar to SVC with parameter kernel='linear',...	it has more flexibility in the choice of pe...	NaN
19	nusvr	nuSVR	Similar to SVR same caveat as above	it has more flexibility in the choice of pe...	NaN
20	pca	Principal Component Analysis (PCA)	Dimensionality reduction. Pre processto reduce...	PCA can help us improve performance at a very ...	intel sklearnex: All parameters except svd_sol...
21	random_forest_classifier	Random Forest Classifier	Classifier: A random forest produces good pred...	One third of data is not used for training, he...	Less interpret-ability, black box approach ...
22	random_forest_regressor	Random Forest Regressor	Regressor: A random forest produces good predi...	One third of data is not used for training, he...	Less interpret-ability, black box approach ...
23	randomforestregressor	alias for Random Forest Regressor	same as above	same as above	same as above
24	randomrorestclassifier	alias for Random Forest Classifier	same as above	same as above	same as above
25	ridge	Ridge Regression	Ridge is a linear regression with an L2 regu...	Trades variance for bias (i.e. in presence of ...	Intel sklearnex: \nAll parameters except norma...
26	roc_auc_score	ROC AUC score	Receiver Operator Characteristics (ROC) Area U...	A simple graphical representation of the diagn...	Intel sklearnex: Parameters average, sample_we...
27	set_config	set_config	We will ignore this one for the course, but it...	NaN	NaN
28	svc	Support Vector Classifier	Classification: requires X & y. It is based o...	1. Regularization capabilities: SVM has L2 Reg...	1. Choosing an appropriate Kernel function is ...
29	svr	Support Vector Regressor	Regression: requires X & y. It is based on SVM...	1. Regularization capabilities: SVM has L2 Reg...	1. Choosing an appropriate Kernel function is ...
30	train_test_split	Train Test Split	Data preparation to split data into a training...	Easy to implement and interpret. Less time con...	f the dataset is small, keeping a portion for ...
31	tsne	t-Distributed Stochastic Neighbor Embedding	dimensionality reduction techniques in Machine...	1. Handles Non Linear Data Efficiently: PCA is...	sklearnex must only use 'euclidean' or 'minko...

通过代码查看已优化的算法列表

```
# finding name info can be confusing
# as there are aliases to multiple fuctions
from sklearnex import get_patch_names, get_patch_map
sorted(get_patch_names()) #get_patch_map() for more details

['config_context',
 'dbscan',
 'distances',
 'elasticnet',
 'fin_check',
 'get_config',
 'kmeans',
 'kneighborsclassifier',
 'kneighborsregressor',
 'knn_classifier',
 'knn_regressor',
 'lasso',
 'linear',
 'log_reg',
 'logistic',
 'logisticregression',
 'nearest_neighbors',
 'nearestneighbors',
 'nusvc',
 'nusvr',
 'pca',
 'random_forest_classifier',
 'random_forest_regressor',
 'randomforestregressor',
 'randomrorestclassifier',
 'ridge',
 'roc_auc_score',
 'set_config',
 'svc',
 'svr',
 'train_test_split',
 'tsne']
```

Scikit-learn算法加速-如何为现有代码加速

- 无需修改原有代码的加速方法：

`python -m sklearnex my_application.py`

- Jupyter notebook的加速方法：

`import the sklearnex library`

`patch_sklearn()` #或者指定加速某算法:`patch_sklearn( "SVC" )` 或`patch_sklearn(["SVC", "DBSCAN"])`

在此之后就生效，例如再执行如下代码，就自动加速：

`from sklearnex.neighbors import NearestNeighbors, PCA, Kmeans`

- 移除加速的方法：

先调用：

`unpatch_sklearn()`

在后重新引入相关算法：

`from sklearn.cluster import PCA`

Scikit-learn算法加速测试

- 测试加速效果的性能对比
- 制作一个比较函数，生成每个算法的数据，创建算法的估计器
- 分别计算训练和预测时间
- 定义函数将算法与数据集配对

```
from tqdm import tqdm # progress bar
import time #so we can time the action
def comparison(cases):
    elapsed_fit = {} # dictionary to track the time elapsed for the fit method
    elapsed_predict = {} # dictionary to track the time elapsed for the predict/transform method
    # the parameters for this algorithms and for generating the data will be in the next cell
    for name, case in tqdm(cases.items()):
        algorithm = case['algorithm']
        estimator = algorithm['estimator'](**algorithm['params'])
        data = case['data']
        x, y = data['generator'](**data['params'])

        # Timing of fit and predict
        start = time.time()
        estimator.fit(x, y)
        fit_time = time.time() - start
        start = time.time()
        if hasattr(estimator, 'predict'):
            estimator.predict(x)
        if hasattr(estimator, 'transform'):
            estimator.transform(x)
        predict_time = time.time() - start

        elapsed_fit[name] = fit_time
        elapsed_predict[name] = predict_time
    return elapsed_fit, elapsed_predict
```

elapsed_fit变量中存放算法训练耗时

elapsed_predict变量中存放推理耗时

Scikit-learn算法加速测试

get_cases方法中定义了：

1. 算法的名称
2. 算法的参数项和参数值
3. 对应的数据集（生成器）

通过逐一遍历这些算法，进行性能对比

```
def get_cases():  
    return {  
        'SVC with Linear Kernel': {  
            "algorithm": {  
                'estimator': sklearn.svm.SVC,  
                'params': {  
                    'C': 1.0,  
                    'kernel': 'linear',  
                }  
            },  
            "data": {  
                'generator': sklearn.datasets.make_classification,  
                'params': {  
                    'n_samples': 20000,  
                    'n_features': 30,  
                    'n_classes': 3,  
                    'n_informative': 3,  
                    'random_state': 43,  
                }  
            }  
        },  
        'SVC with RBF Kernel': {  
            "algorithm": {  
                'estimator': sklearn.svm.SVC,  
                'params': {  
                    'C': 1.0,  
                    'kernel': 'rbf',  
                }  
            },  
            "data": {  
                'generator': sklearn.datasets.make_classification,  
                'params': {  
                    'n_samples': 25000,  
                    'n_features': 30,  
                    'n_classes': 5,  
                    'n_informative': 4,  
                    'random_state': 43,  
                }  
            }  
        }  
    }
```

Scikit-learn算法加速测试-原算法运行耗时

先看未加速前的8个算法的运行耗时：

```
from sklearnx import patch_sklearn, unpatch_sklearn
unpatch_sklearn() # this will set parameters such that the stock version of sklearn will be called
import sklearn.svm, sklearn.datasets, sklearn.neighbors, sklearn.linear_model, sklearn.decomposition
cases = get_cases() #case the algorithm/dataset pairs
sklearn_fit, sklearn_predict = comparison(cases) # call the comparison function to captures the elapsed time dictio
```

```
25%|███████| 2/8 [01:39<04:52, 48.79s/it] Intel(R) Extension for Scikit-learn* enabled (https://github.com/intel/scikit-learn-intelex)
100%|██████████| 8/8 [04:19<00:00, 32.43s/it]
```

```
print('Result of scikit-learn train: ', sklearn_fit)
print('\n')
print('Result of scikit-learn predict: ', sklearn_predict)
```

```
Result of scikit-learn train: {'SVC with Linear Kernel': 45.61663222312927, 'SVC with RBF Kernel': 22.4045672416687, 'Logistic Regression': 13.256971597671509, 'KNN Classifier': 0.004821062088012695, 'KNN Regression': 0.0013971328735351562, 'Linear Regression': 13.063661813735962, 'Ridge Regression': 6.973957777023315, 'PCA': 4.730371952056885}
```

```
Result of scikit-learn predict: {'SVC with Linear Kernel': 8.172053098678589, 'SVC with RBF Kernel': 22.623200178146362, 'Logistic Regression': 0.16436409950256348, 'KNN Classifier': 23.463303565979004, 'KNN Regression': 24.288088083267212, 'Linear Regression': 0.9629719257354736, 'Ridge Regression': 0.7868826389312744, 'PCA': 0.5773224830627441}
```

可以看到线性SVC耗时较长

Scikit-learn算法加速测试-加速后算法运行耗时

先看加速后的8个算法的运行耗时：

```
from sklearnx import patch_sklearn, unpatch_sklearn
# insert code ( a single line) to patch sklearn here:
patch_sklearn()
```

```
import sklearn.svm, sklearn.datasets, sklearn.neighbors, sklearn.linear_model, sklearn.decomposition
cases = get_cases()
sklearnx_fit, sklearnx_predict = comparison(cases)
```

```
Intel(R) Extension for Scikit-learn* enabled (https://github.com/intel/scikit-learn-intelx)
100%|██████████| 8/8 [01:20<00:00, 10.11s/it]
```

```
print('Result of scikit-learn* train: ', sklearnx_fit)
print('\n')
print('Result of scikit-learn* predict: ', sklearnx_predict)
```

```
Result of scikit-learn* train: {'SVC with Linear Kernel': 2.0269083976745605, 'SVC with RBF Kernel': 1.718588352203
3691, 'Logistic Regression': 1.0187699794769287, 'KNN Classifier': 0.0794711129760742, 'KNN Regression': 0.00234889
98413085938, 'Linear Regression': 0.5390632152557373, 'Ridge Regression': 0.6380298137664795, 'PCA': 0.3723585605621
338}
```

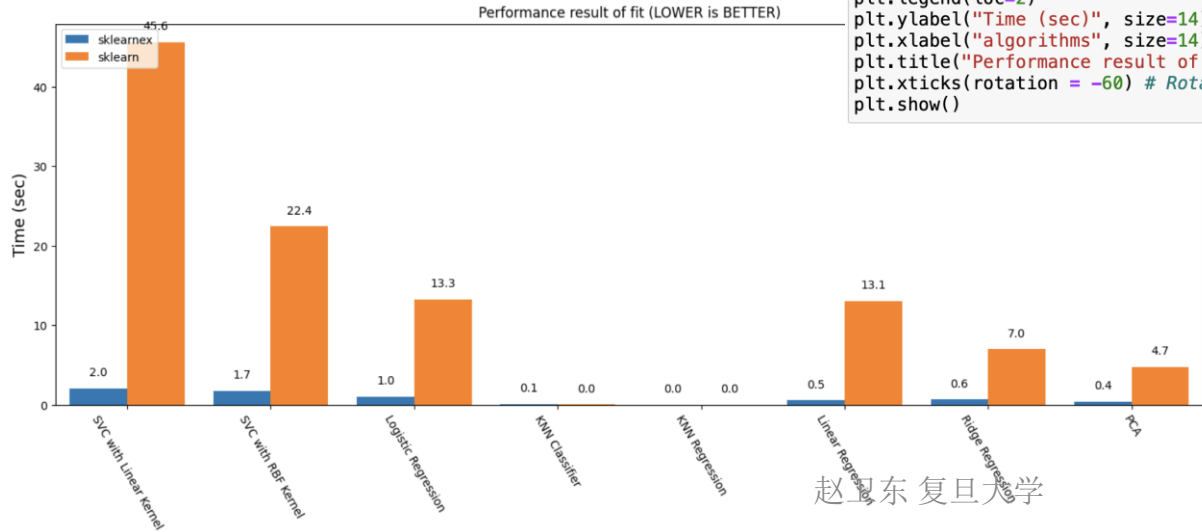
```
Result of scikit-learn* predict: {'SVC with Linear Kernel': 0.26720738410949707, 'SVC with RBF Kernel': 0.450794935
22644043, 'Logistic Regression': 0.07174301147460938, 'KNN Classifier': 0.29738306999206543, 'KNN Regression': 0.304
3701648712158, 'Linear Regression': 0.35891032218933105, 'Ridge Regression': 0.23721861839294434, 'PCA': 0.103921175
00305176}
```

可以看到原来耗时较长的线性SVC，只需2秒训练，0.26秒推理预测

Scikit-learn算法加速测试-训练加速前后算法耗时对比

训练加速前后实验对比:

- sklearn表示加速前
- sklearnex表示加速后



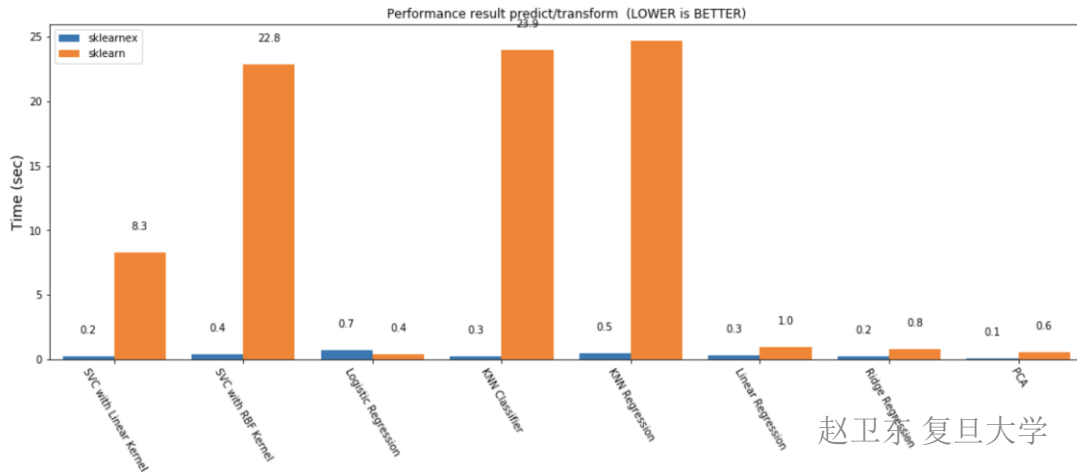
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
result = { name_sk: [time_ex, time_sk] for (name_sk, time_sk),
           (name_ex, time_ex) in zip(sklearn_fit.items(), sklearnex_fit.items())}
result['Library'] = ['sklearnex', 'sklearn']
df_fit = pd.DataFrame(data=result).melt('Library', var_name='Algorithms', value_name='Time')
fig = plt.figure(figsize=(18, 6))
fig.patch.set_alpha(1)
plt.subplot(1, 1, 1)
barplot = sns.barplot(x='Algorithms', y='Time', hue='Library', data=df_fit, errwidth = 2,
                      capsize = 0.05, saturation = 8,)
for p in barplot.patches:
    barplot.annotate(format(p.get_height(), '.1f'),
                     (p.get_x() + p.get_width() / 2., p.get_height() + 2),
                     ha = 'center', va = 'center')
plt.legend(loc=2)
plt.ylabel("Time (sec)", size=14)
plt.xlabel("algorithms", size=14)
plt.title("Performance result of fit (LOWER is BETTER)")
plt.xticks(rotation = -60) # Rotates X-Axis Ticks by 60-degrees
plt.show()
```

Scikit-learn算法加速测试-推理加速前后算法耗时对比

推理加速前后实验对比:

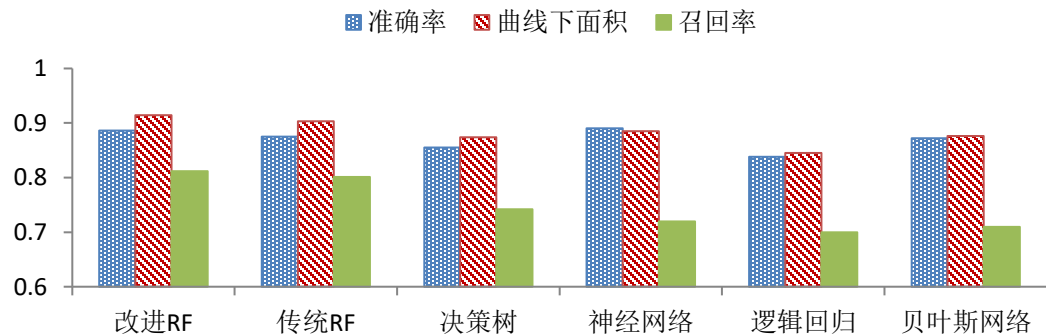
- sklearn表示加速前
- sklearnex表示加速后

```
result = { name_sk: [time_ex, time_sk] for (name_sk, time_sk),
          (name_ex, time_ex) in zip(sklearn_predict.items(), sklearnex_predict.items())}
result['Library'] = ['sklearnex', 'sklearn']
df_predict = pd.DataFrame(data=result).melt('Library', var_name='Algorithms', value_name='Time')
fig = plt.figure(figsize=(18, 6))
fig.patch.set_alpha(1)
plt.subplot(1, 1, 1)
barplot = sns.barplot(x='Algorithms', y='Time', hue='Library', data=df_predict, errwidth = 2,
                      capsize = 0.05, saturation = 8,)
for p in barplot.patches:
    barplot.annotate(format(p.get_height(), '.1f'), (p.get_x() + p.get_width() / 2., p.get_height() + 2),
                     ha = 'center', va = 'center')
plt.legend(loc=2)
plt.ylabel("Time (sec)", size=14)
plt.xlabel("algorithms", size=14)
plt.title("Performance result predict/transform (LOWER is BETTER)")
plt.xticks(rotation = -60) # Rotates X-Axis Ticks by 60-degrees
plt.show()
```



实际项目的验证

集装箱危险品瞒报预测



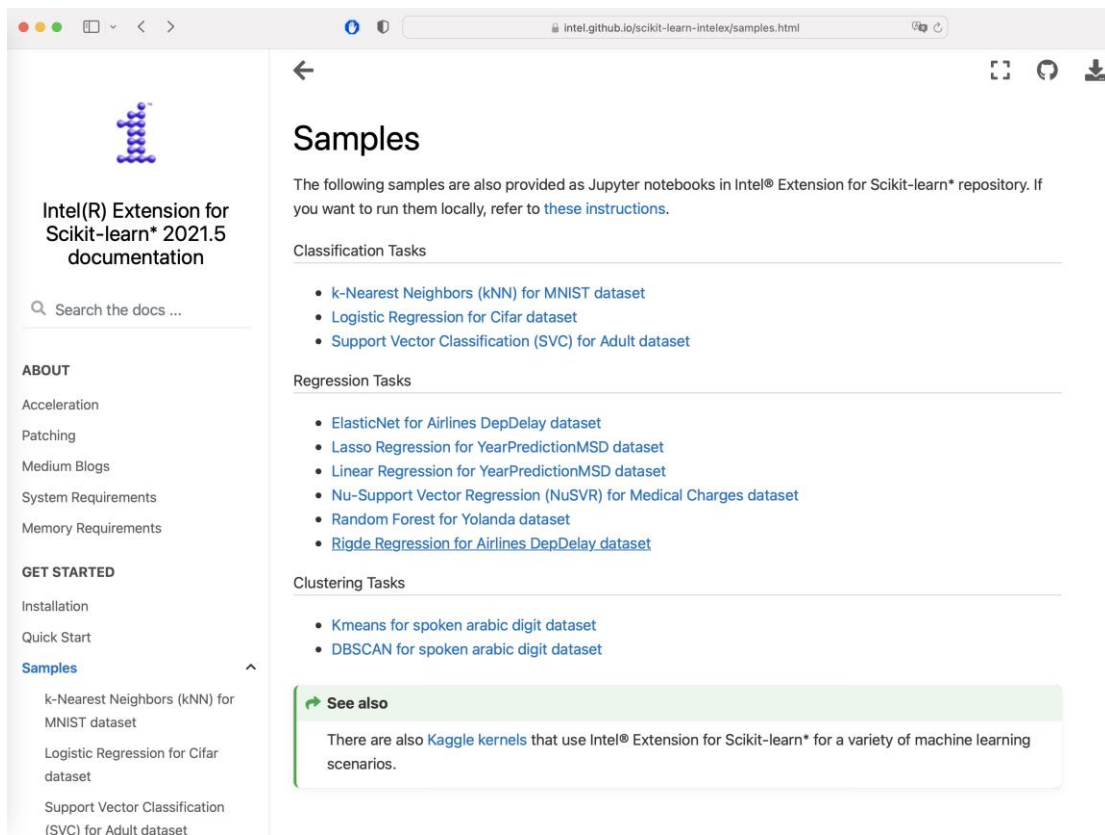
	stock		intel	perf gain
read_excel	31.09	read_csv	3.63	8.564738
pre_process	36.7		45.91	
decisionTreeClf			1596.98	
mlp	4837		437.9	11.0459
lr	1990.26		120.91	16.46067
oldRandomForestClf	461.27		118.65	3.887653
randomForestClf	458.39		122.22	3.750532
nbClf	119.13		45.97	2.591473
OMP_NUM_THREADS=56 numactl -C 0-55 -p 1 python main.py				

在DevCloud使用JupyterLab进行实验

相关源码和可参考文档:

<https://github.com/intel/scikit-learn-intelex>

- The software acceleration provided by Intel® Extension for Scikit-learn* is achieved through the use of vector instructions, IA hardware-specific memory optimizations, threading, and optimizations for all upcoming Intel platforms at launch time.
- Intel® Extension for Scikit-learn* dynamically patches scikit-learn estimators to use Intel(R) oneAPI Data Analytics Library as the underlying solver, while getting the same solution faster.



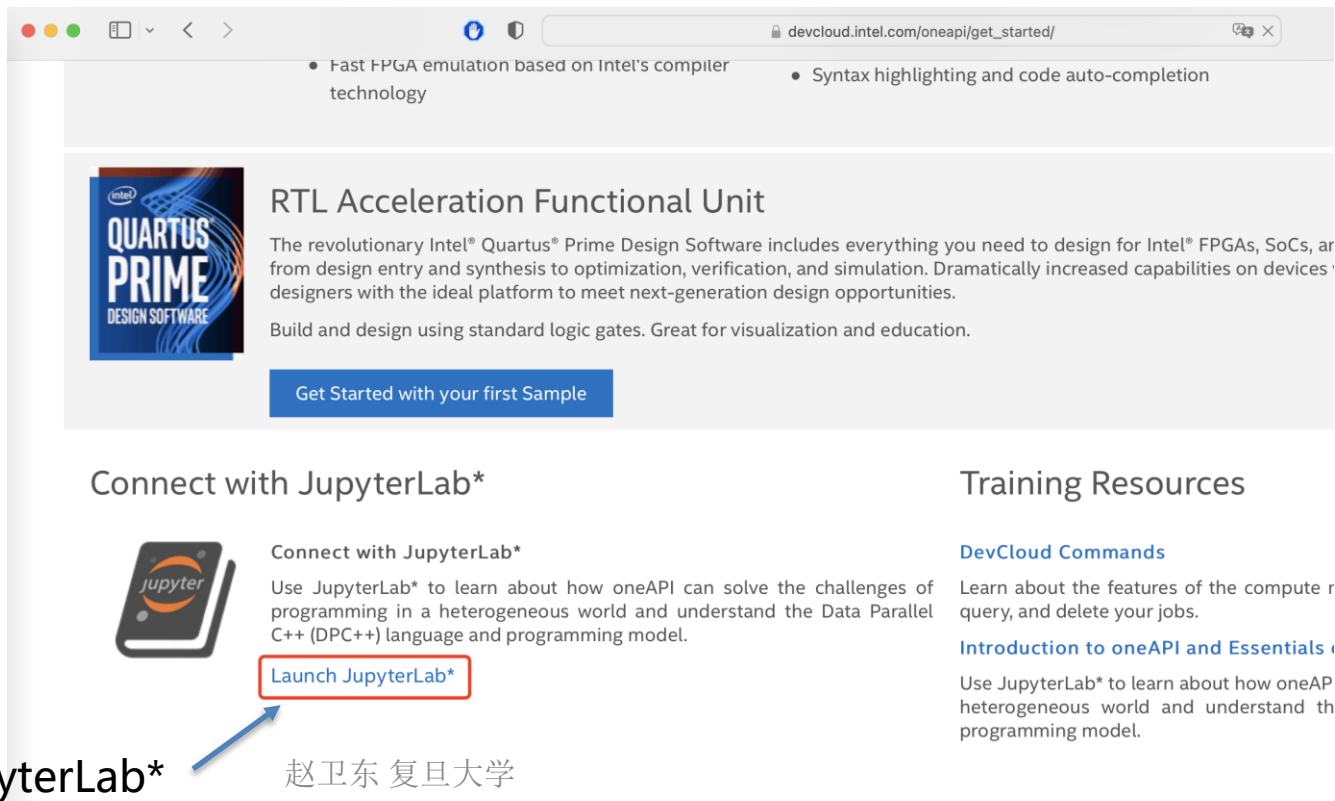
在DevCloud使用JupyterLab进行实验

可亲身基于 DevCloud实操Intel oneAPI Toolkits

使用 JupyterLab* 可在线
实践了解 oneAPI 如何解
决异构世界中的编程挑战

内含较多代码示例和文档
供学习和练习

注册并点击这里启动JupyterLab*



• Fast FPGA emulation based on Intel's compiler technology

• Syntax highlighting and code auto-completion

RTL Acceleration Functional Unit

The revolutionary Intel® Quartus® Prime Design Software includes everything you need to design for Intel® FPGAs, SoCs, and more, from design entry and synthesis to optimization, verification, and simulation. Dramatically increased capabilities on devices, giving designers the ideal platform to meet next-generation design opportunities.

Build and design using standard logic gates. Great for visualization and education.

[Get Started with your first Sample](#)

Connect with JupyterLab*



Connect with JupyterLab*

Use JupyterLab* to learn about how oneAPI can solve the challenges of programming in a heterogeneous world and understand the Data Parallel C++ (DPC++) language and programming model.

[Launch JupyterLab*](#)

Training Resources

[DevCloud Commands](#)

Learn about the features of the compute environment, query, and delete your jobs.

[Introduction to oneAPI and Essentials](#)

Use JupyterLab* to learn about how oneAPI can solve the challenges of programming in a heterogeneous world and understand the Data Parallel C++ (DPC++) language and programming model.

赵卫东 复旦大学

