Jupyter Notebook魔法(Magic)命令

机器学习研究会订阅号 10月26日

IPython提供了许多魔法命令，使得在IPython环境中的操作更加得心应手。魔法命令都以%或者%%开头，以%开头的成为行命令，%%开头的称为单元命令。行命令只对命令所在的行有效，而单元命令则必须出现在单元的第一行，对整个单元的代码进行处理。

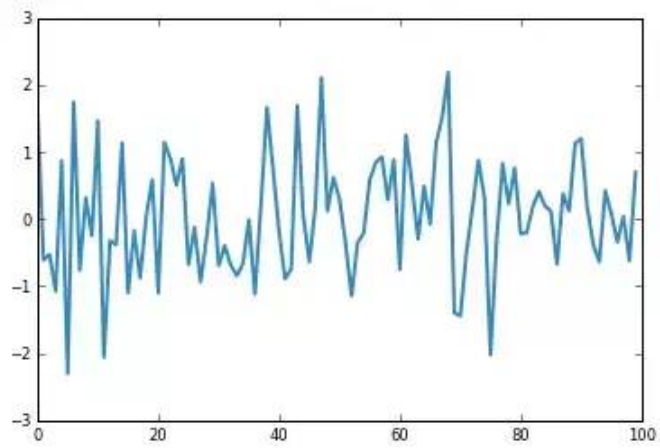
执行%magic可以查看关于各个命令的说明，而在命令之后添加?可以查看该命令的详细说明。

显示**matplotlib**图表

matplotlib是最著名的Python图表绘制扩展库，它支持输出多种格式的图形图像，并且可以使用多种GUI界面库交互式地显示图表。使用%matplotlib命令可以将matplotlib的图表直接嵌入到Notebook之中，或者使用指定的界面库显示图表，它有一个参数指定matplotlib图表的显示方式。

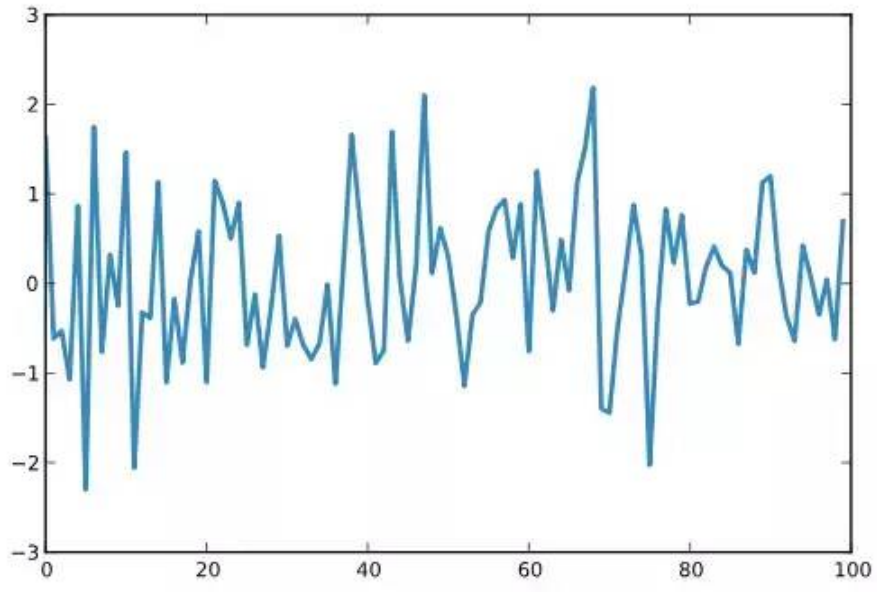
在下面的例子中，inline表示将图表嵌入到Notebook中。因此最后一行pl.plot()所创建的图表将直接显示在该单元之下，由于我们不需要查看最后一行返回的对象，因此以分号结束该行。

```
%matplotlib inlineimport pylab as pl
pl.seed(1)
data = pl.randn(100)
pl.plot(data);
```



内嵌图表的输出格式缺省为PNG，可以通过%config命令修改这个配置。%config命令可以配置IPython中的各个可配置对象，其中InlineBackend对象为matplotlib输出内嵌图表时所使用的对象，我们配置它的figure_format="svg"，这样将内嵌图表的输出格式修改为SVG。

```
%config InlineBackend.figure_format="svg"%matplotlib inlinepl.plot(data);
```



内嵌图表很适合制作图文并茂的Notebook，然而它们是静态的无法进行交互。这时可以将图表输出模式修改为使用GUI界面库，下面的qt4表示使用QT4界面库显示图表。请读者根据自己系统的配置，选择合适的界面库：'gtk', 'osx', 'qt', qt4', 'tk', 'wx'。

执行下面的语句将弹出一个窗口显示图表，可以通过鼠标和键盘与此图表交互。请注意该功能只能在运行IPython Kernel的机器上显示图表。

```
%matplotlib qt4pl.plot(data);
```

性能分析

性能分析对编写处理大量数据的程序非常重要，特别是Python这样的动态语言，一条语句可能会执行很多内容，有的是动态的，有的调用二进制扩展库，不进行性能分析，就无法对程序进行优化。IPython提供了许多进行性能分析的魔法命令。

%timeit调用timeit模块对单行语句重复执行多次，计算出其执行时间。下面的代码测试修改列表单个元素所需的时间。

```
a = [1,2,3]
%timeit a[1] = 1010000000 loops, best of 3: 164 ns per loop
```

%%timeit则用于测试整个单元中代码的执行时间。下面的代码测试空列表中循环添加10个元素所许的时间：

```
%%timeit
a = []for i in xrange(10):
  a.append(i)100000 loops, best of 3: 3.85 µs per loop
```

timeit命令会重复执行代码多次，而time则只执行一次代码，输出代码的执行情况，和timeit命令一样，它可以作为行命令和单元命令。下面的代码统计往空列表中添加10万个元素所需的时间。

```
%%time
a = []for i in xrange(100000):
  a.append(i)
CPU times: user 44 ms, sys: 4 ms, total: 48 ms
Wall time: 51.9 ms
```

time和timeit命令都将信息使用print输出，如果希望用程序分析这些信息，可以使用%%capture命令，将单元格的输出保存为一个对象。下面的程序对不同长度的数组调用sort()函数进行排序，并使用%timeit命令统计排序所需的时间。为了加快程序的计算速度，这里通过-n20指定代码的运行次数为20次。由于使用了%%capture命令，程序执行之后没有输出，所有输出都被保存进了result对象。

```
%%capture resultimport numpy as npfor n in [1000, 5000, 10000, 50000, 100000, 500000]:
    arr = np.random.rand(n)
print "n={0}".format(n)
    %timeit -n20 np.sort(arr)
```

result.stdout属性中保存通过标准输出管道中的输出信息：

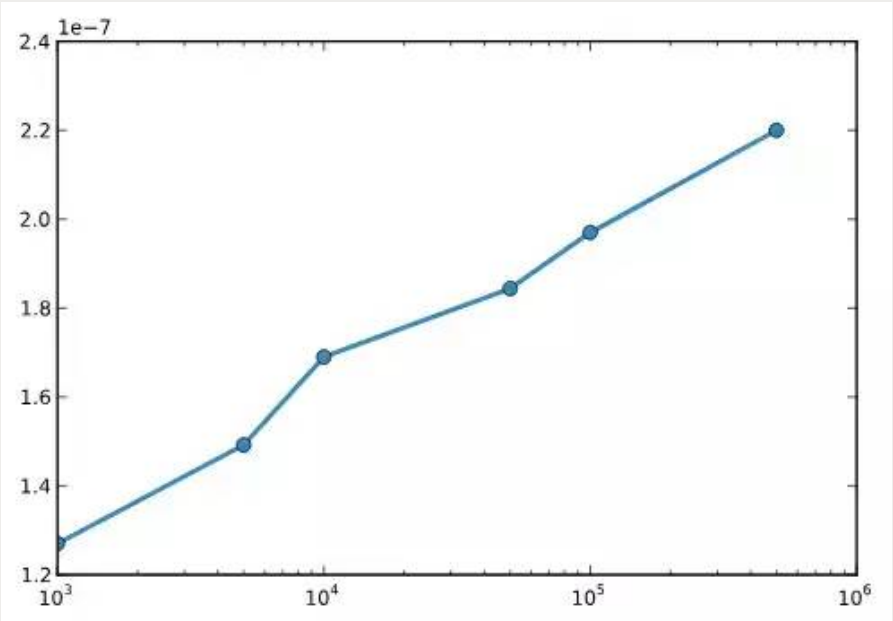
```
print result.stdout
n=100020 loops, best of 3: 127 us per loop
n=500020 loops, best of 3: 746 us per loop
n=1000020 loops, best of 3: 1.69 ms per loop
n=5000020 loops, best of 3: 9.22 ms per loop
n=10000020 loops, best of 3: 19.7 ms per loop
n=50000020 loops, best of 3: 110 ms per loop
```

下面的代码使用re模块从上面的字符串中获取数组长度和排序执行时间的信息，并将其绘制成图表。图表的横坐标为对数坐标轴，表示数组的长度；纵坐标为平均每个元素所需的排序时间。可以看出每个元素所需的平均排序时间与数组的长度的对数成正比，因此可以计算出排序函数sort()的时间复杂度为：

$$O(n \log n)$$

```
def tosec(t):
    units = {"ns":1e-9, "us":1e-6, "ms":1e-3, "s":1}
    value, unit = t.strip().split()    return float(value) * units[unit]import re
info = re.findall(r"n=(.+?)\n.+?best of 3: (.+?) per loop", result.stdout)
```

```
info = [(int(t0), tosec(t1)) for t0, t1 in info]
x, y = np.r_[info].T
pl.semilogx(x, y/x, "-o");
```



%%%prun命令调用profile模块，对单元中的代码进行性能剖析。下面的性能剖析显示fib()运行了21891次，而fib_fast()则只运行了20次。

```
%%nopage
%%prundef fib(n):
if n < 2:
  return 1
else:
  return fib(n-1) + fib(n-2)

def fib_fast(n, a=1, b=1):

  if n == 1:
    return b
  else:
    return fib_fast(n-1, b, a+b)

fib(20)
fib_fast(20)

21913 function calls (4 primitive calls) in 0.084 seconds

Ordered by: internal time


ncalls  tottime  percall  cumtime  percall filename:lineno(function)21891/1    0.084    0.000    0.084

1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profi
```

- END -

想要了解更多资讯，请扫描下方二维码，关注机器学习研究会

转自： 人工智能