Donnerstag: Python für Machine Learning

9 Python für Machine Learning
- Profiling
- Numpy und SciPy
- scikit-learn

Python für Machine Learning

Profiling - Warum?[1]

- Viele Ursachen sind möglich, wenn ein Programm langsam läuft.
- Zeitaufwand minimieren durch Optimierung an der richtigen Stelle!
- Beim Profiling misst man den Ressourcenverbrauch eines Programms (Berechnungszeit, RAM).
- Faustregel: Erst profilen, dann optimieren.
  - Wie lange läuft das Programm?
  - Wie oft wird eine Funktion/Zeile ausgeführt?
  - Wieviel Speicher benötigt eine Datenstruktur?
- Optimierung an der falschen Stelle $\rightarrow$ großer Aufwand, kaum Verbesserungen

[1] Slides basieren auf Gorelick, Micha und Ozsvald, Ian: *High performance Python*. O'Reilly, 2014.

Profiling - Warum?[1]

- Viele Ursachen sind möglich, wenn ein Programm langsam läuft.
- Zeitaufwand minimieren durch Optimierung an der richtigen Stelle!
- Beim Profiling misst man den Ressourcenverbrauch eines Programms (Berechnungszeit, RAM).
- Faustregel: Erst profilen, dann optimieren.
  - Wie lange läuft das Programm?
  - Wie oft wird eine Funktion/Zeile ausgeführt?
  - Wieviel Speicher benötigt eine Datenstruktur?
- Optimierung an der falschen Stelle $\rightarrow$ großer Aufwand, kaum Verbesserungen

[1] Slides basieren auf Gorelick, Micha und Ozsvald, Ian: *High performance Python*. O'Reilly, 2014.

Profiling - Warum?[1]

- Viele Ursachen sind möglich, wenn ein Programm langsam läuft.
- Zeitaufwand minimieren durch Optimierung an der richtigen Stelle!
- Beim Profiling misst man den Ressourcenverbrauch eines Programms (Berechnungszeit, RAM).
- Faustregel: Erst profilen, dann optimieren.
  - Wie lange läuft das Programm?
  - Wie oft wird eine Funktion/Zeile ausgeführt?
  - Wieviel Speicher benötigt eine Datenstruktur?
- Optimierung an der falschen Stelle $\rightarrow$ großer Aufwand, kaum Verbesserungen

[1] Slides basieren auf Gorelick, Micha und Ozsvald, Ian: *High performance Python*. O'Reilly, 2014.

Profiling - Warum?[1]

- Viele Ursachen sind möglich, wenn ein Programm langsam läuft.
- Zeitaufwand minimieren durch Optimierung an der richtigen Stelle!
- Beim Profiling misst man den Ressourcenverbrauch eines Programms (Berechnungszeit, RAM).
- Faustregel: Erst profilen, dann optimieren.
  - Wie lange läuft das Programm?
  - Wie oft wird eine Funktion/Zeile ausgeführt?
  - Wieviel Speicher benötigt eine Datenstruktur?
- Optimierung an der falschen Stelle $\rightarrow$ großer Aufwand, kaum Verbesserungen

[1] Slides basieren auf Gorelick, Micha und Ozsvald, Ian: *High performance Python*. O'Reilly, 2014.

Profiling - Warum?[1]

- Viele Ursachen sind möglich, wenn ein Programm langsam läuft.
- Zeitaufwand minimieren durch Optimierung an der richtigen Stelle!
- Beim Profiling misst man den Ressourcenverbrauch eines Programms (Berechnungszeit, RAM).
- Faustregel: Erst profilen, dann optimieren.
  - Wie lange läuft das Programm?
  - Wie oft wird eine Funktion/Zeile ausgeführt?
  - Wieviel Speicher benötigt eine Datenstruktur?
- Optimierung an der falschen Stelle $\rightarrow$ großer Aufwand, kaum Verbesserungen

[1] Slides basieren auf Gorelick, Micha und Ozsvald, Ian: *High performance Python*. O'Reilly, 2014.

Profiling - Warum?[1]

- Viele Ursachen sind möglich, wenn ein Programm langsam läuft.
- Zeitaufwand minimieren durch Optimierung an der richtigen Stelle!
- Beim Profiling misst man den Ressourcenverbrauch eines Programms (Berechnungszeit, RAM).
- Faustregel: Erst profilen, dann optimieren.
  - Wie lange läuft das Programm?
  - Wie oft wird eine Funktion/Zeile ausgeführt?
  - Wieviel Speicher benötigt eine Datenstruktur?
- Optimierung an der falschen Stelle $\rightarrow$ großer Aufwand, kaum Verbesserungen

[1] Slides basieren auf Gorelick, Micha und Ozsvald, Ian: *High performance Python*. O'Reilly, 2014.

Profiling - Warum?[1]

- Viele Ursachen sind möglich, wenn ein Programm langsam läuft.
- Zeitaufwand minimieren durch Optimierung an der richtigen Stelle!
- Beim Profiling misst man den Ressourcenverbrauch eines Programms (Berechnungszeit, RAM).
- Faustregel: Erst profilen, dann optimieren.
  - Wie lange läuft das Programm?
  - Wie oft wird eine Funktion/Zeile ausgeführt?
  - Wieviel Speicher benötigt eine Datenstruktur?
- Optimierung an der falschen Stelle $\rightarrow$ großer Aufwand, kaum Verbesserungen

[1] Slides basieren auf Gorelick, Micha und Ozsvald, Ian: *High performance Python*. O'Reilly, 2014.

Profiling - Warum?[1]

- Viele Ursachen sind möglich, wenn ein Programm langsam läuft.
- Zeitaufwand minimieren durch Optimierung an der richtigen Stelle!
- Beim Profiling misst man den Ressourcenverbrauch eines Programms (Berechnungszeit, RAM).
- Faustregel: Erst profilen, dann optimieren.
  - Wie lange läuft das Programm?
  - Wie oft wird eine Funktion/Zeile ausgeführt?
  - Wieviel Speicher benötigt eine Datenstruktur?
- Optimierung an der falschen Stelle $\rightarrow$ großer Aufwand, kaum Verbesserungen

---

[1] Slides basieren auf Gorelick, Micha und Ozsvald, Ian: *High performance Python*. O'Reilly, 2014.

Modul `timeit`

http://docs.python.org/library/timeit.html

- Modul, um kleine Pythoncodestücke zu timen.
- Handhabung im Interpreter:

```
>>> import timeit
>>> setup = "l = range(10000)" # parameters
>>> stmt = "len(l)" # statement to evaluate
>>> timeit.timeit(stmt=stmt, setup=setup, number=10000)
0.0019829273223876953
```

- `stmt`: Ausdruck, dessen benötigte Zeit gemessen werden soll
- `setup`: Anweisungen, die (einmal) vor der Ausführung von `stmt` durchgeführt werden sollen
- `number`: Anzahl der Ausführungen von `stmt` (Default 1000000)

Rückgabe: Benötigte Zeit in Sekunden

Modul cProfile

https://docs.python.org/2/library/profile.html

- `timeit` ist sehr umständlich, wenn man Funktionen oder Programme evaluieren will.
- `cProfile` ist ein in der Standardbibliothek eingebautes Werkzeug, das misst, wie oft und wie lange Teile des Programms ausgeführt wurden.
- Warnung: Profiling verlängert die Laufzeit
- Profiler auf ein Skript anwenden:

```
$ python -m cProfile myProgram.py
```

Beispiel: Tiny Search Engine

Idee

- Implementierung einer Mini-Suchmaschine
- Die Suchanfrage (*query*) und die Dokumente (*documents*) sind Worthäufigkeitsvektoren.
- Ähnlichkeit zwischen Suchanfrage und Dokument: Kosinus

Tiny Search Engine: Pseudodaten

Anstatt echte Dokumente einzulesen, werden für Suchanfrage und Dokumente zufällig “Termhäufigkeiten” zwischen 0 und 100 gezogen:

```
def pseudo_data(l):
    v = []
    for i in range(l):
        freq = random.randint(0, 100)
        v.append(freq)
```

Tiny Search Engine: Kosinusähnlichkeit

$$\cos(a, b) = \frac{a \cdot b}{\|a\|\|b\|} = \frac{a \cdot b}{\sqrt{\sum_i a_i^2}\sqrt{\sum_i b_i^2}}$$

Einfache Python-Implementierung

```
def cosine(a, b):
    assert(len(a)==len(b))
    ab_sum = 0
    a_sq_sum = 0
    b_sq_sum = 0
    for i in range(len(a)):
        ab_sum += a[i] * b[i]
        a_sq_sum += a[i] * a[i]
        b_sq_sum += b[i] * b[i]
    total = ab_sum / (math.sqrt(a_sq_sum) * math.sqrt(b_sq_sum))
    return total
```

Tiny Search Engine: run

```
def run(vocab_size, ndocs):
    random.seed()
    query = pseudo_data(vocab_size)
    maxsim = 0
    best = []
    for i in range(ndocs):
        doc = pseudo_data(vocab_size)
        cos = cs.cosine(query, doc)
        if cos > maxsim:
            maxsim = cos
            best = doc

    return (maxsim, best)
```

cProfile Aufruf

```
$ python -m cProfile -s cumtime \
tiny_search_engine.py --vocab-size 10000 \
--num-docs 1000
```

`-s`: Sortierung der Ausgabe (hier nach kumulativer Laufzeit)

Ausgegebene Statistiken (in Spalten)

ncalls	Anzahl der Aufrufe
tottime	Gesamtlaufzeit dieser Funktion (ohne Subfunktionen)
percall	tottime / ncalls
cumtime	Gesamtlaufzeit dieser Funktion (mit Subfunktionen)
percall	cumtime / ncalls

cProfile Aufruf

```
$ python -m cProfile -s cumtime \
tiny_search_engine.py --vocab-size 10000 \
--num-docs 1000
```

`-s`: Sortierung der Ausgabe (hier nach kumulativer Laufzeit)

Ausgegebene Statistiken (in Spalten)

ncalls	Anzahl der Aufrufe
tottime	Gesamtlaufzeit dieser Funktion (ohne Subfunktionen)
percall	tottime / ncalls
cumtime	Gesamtlaufzeit dieser Funktion (mit Subfunktionen)
percall	cumtime / ncalls

cProfile Ausgabe

```
         40092169 function calls (40091230 primitive calls) in 19.384 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   19.385   19.385 tiny_search_engine.py:3(<module>)
        1    0.000    0.000   19.282   19.282 tiny_search_engine.py:109(main)
        1    0.016    0.016   19.281   19.281 tiny_search_engine.py:44(run)
     1001    3.987    0.004   17.195    0.017 tiny_search_engine.py:33(pseudo_data)
 10010000    3.300    0.000   12.412    0.000 random.py:238(randint)
 10010000    8.321    0.000    9.111    0.000 random.py:175(randrange)
     1000    2.012    0.002    2.070    0.002 cosinus.py:8(cosine)
 10010000    0.791    0.000    0.791    0.000 {method 'random' of '_random.Random' objects}
 10015034    0.743    0.000    0.743    0.000 {method 'append' of 'list' objects}
     2113    0.111    0.000    0.111    0.000 {range}
```

...

Visualisierung der Profiler-Ausabe

1. Profiler-Ausgabe in Datei schreiben:

```
$ python -m cProfile -o profile.stats
tiny_search_engine.py -v 10000 -d 1000
```

2. gprof2dot: Python-Skript, um Profiler-Ausgabe in dot-Format zu übertragen.[2]

```
$ gprof2dot profile.stats -f pstats > profile.dot
```

3. dot: Tool zum Zeichnen von gerichteten Graphen

```
$ dot profile.dot -Tpng -o profile.png
```

Andere Pakete für Visualisierung: z.B. `snakeviz`, `pycallgraph`

---

[2] siehe `https://github.com/jrfonseca/gprof2dot`. Mehr zum dot-Format: `http://www.graphviz.org/doc/info/lang.html`

Visualisierung der Profiler-Ausabe

1. Profiler-Ausgabe in Datei schreiben:

```
$ python -m cProfile -o profile.stats
tiny_search_engine.py -v 10000 -d 1000
```

2. `gprof2dot`: Python-Skript, um Profiler-Ausgabe in dot-Format zu übertragen.[2]

```
$ gprof2dot profile.stats -f pstats > profile.dot
```

3. `dot`: Tool zum Zeichnen von gerichteten Graphen

```
$ dot profile.dot -Tpng -o profile.png
```

Andere Pakete für Visualisierung: z.B. `snakeviz`, `pycallgraph`

[2] siehe `https://github.com/jrfonseca/gprof2dot`. Mehr zum dot-Format: `http://www.graphviz.org/doc/info/lang.html`

Visualisierung der Profiler-Ausabe

1. Profiler-Ausgabe in Datei schreiben:

```
$ python -m cProfile -o profile.stats
tiny_search_engine.py -v 10000 -d 1000
```

2. `gprof2dot`: Python-Skript, um Profiler-Ausgabe in dot-Format zu übertragen.[2]

```
$ gprof2dot profile.stats -f pstats > profile.dot
```

3. `dot`: Tool zum Zeichnen von gerichteten Graphen

```
$ dot profile.dot -Tpng -o profile.png
```

Andere Pakete für Visualisierung: z.B. `snakeviz`, `pycallgraph`

[2] siehe `https://github.com/jrfonseca/gprof2dot`. Mehr zum dot-Format: `http://www.graphviz.org/doc/info/lang.html`

Visualisierung der Profiler-Ausabe

1. Profiler-Ausgabe in Datei schreiben:

```
$ python -m cProfile -o profile.stats
tiny_search_engine.py -v 10000 -d 1000
```

2. `gprof2dot`: Python-Skript, um Profiler-Ausgabe in dot-Format zu übertragen.[2]

```
$ gprof2dot profile.stats -f pstats > profile.dot
```

3. `dot`: Tool zum Zeichnen von gerichteten Graphen

```
$ dot profile.dot -Tpng -o profile.png
```

Andere Pakete für Visualisierung: z.B. `snakeviz`, `pycallgraph`

---

[2] siehe `https://github.com/jrfonseca/gprof2dot`. Mehr zum dot-Format: `http://www.graphviz.org/doc/info/lang.html`

Callgraph

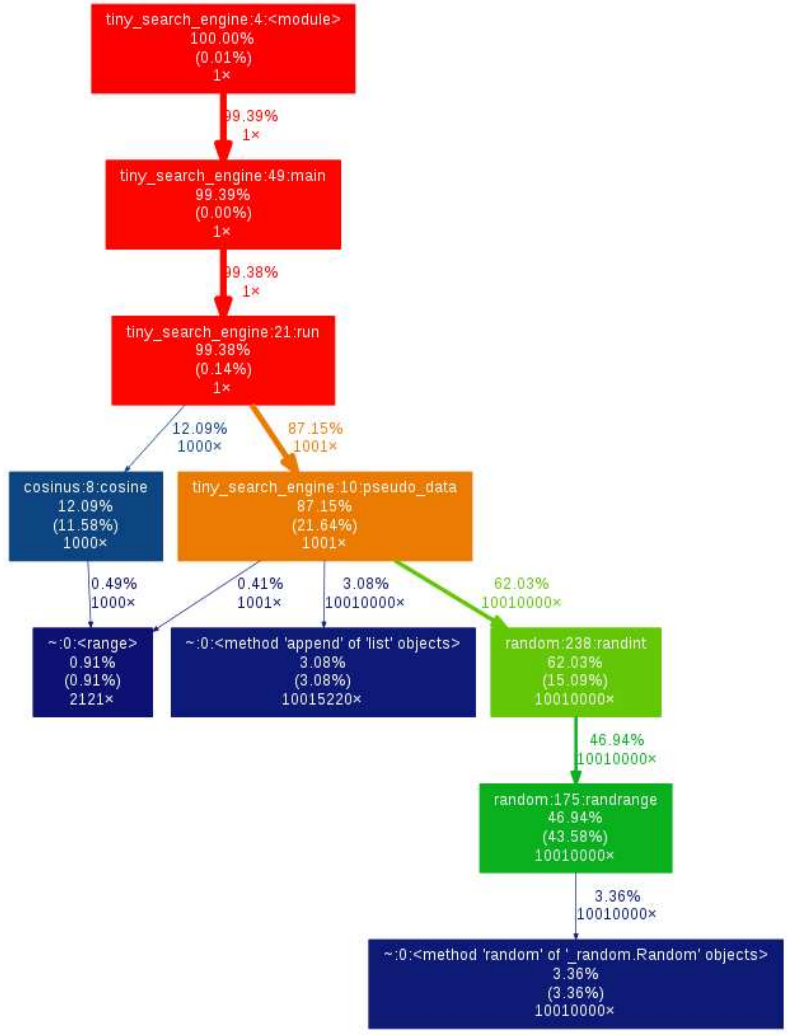

Abbildung: Callgraph für `tiny_search_engine.py`

Zeilenweises Profiling

- `cProfile` gibt Anzahl der Funktionsaufrufe und Laufzeit jeder Funktion an.
- Keine Auskunft darüber, welche Zeile wieviel Zeit benötigt.
- Lösung: `line_profiler` ermöglicht zeilenweises Profil.
- Skript zum Aufruf von `line_profiler`: `kernprof.py`
- Autor: Robert Kern, URL: `https://github.com/rkern/line_profiler`
- Installation: `sudo pip install line_profiler`

Zeilenweises Profiling

- `cProfile` gibt Anzahl der Funktionsaufrufe und Laufzeit jeder Funktion an.
- Keine Auskunft darüber, welche Zeile wieviel Zeit benötigt.
- Lösung: `line_profiler` ermöglicht zeilenweises Profil.
- Skript zum Aufruf von `line_profiler`: `kernprof.py`
- Autor: Robert Kern, URL: `https://github.com/rkern/line_profiler`
- Installation: `sudo pip install line_profiler`

Zeilenweises Profiling

- `cProfile` gibt Anzahl der Funktionsaufrufe und Laufzeit jeder Funktion an.
- Keine Auskunft darüber, welche Zeile wieviel Zeit benötigt.
- Lösung: `line_profiler` ermöglicht zeilenweises Profil.
- Skript zum Aufruf von `line_profiler`: `kernprof.py`
- Autor: Robert Kern, URL: `https://github.com/rkern/line_profiler`
- Installation: `sudo pip install line_profiler`

Zeilenweises Profiling

- `cProfile` gibt Anzahl der Funktionsaufrufe und Laufzeit jeder Funktion an.
- Keine Auskunft darüber, welche Zeile wieviel Zeit benötigt.
- Lösung: `line_profiler` ermöglicht zeilenweises Profil.
- Skript zum Aufruf von `line_profiler`: `kernprof.py`
- Autor: Robert Kern, URL: `https://github.com/rkern/line_profiler`
- Installation: `sudo pip install line_profiler`

Zeilenweises Profiling

- `cProfile` gibt Anzahl der Funktionsaufrufe und Laufzeit jeder Funktion an.
- Keine Auskunft darüber, welche Zeile wieviel Zeit benötigt.
- Lösung: `line_profiler` ermöglicht zeilenweises Profil.
- Skript zum Aufruf von `line_profiler`: `kernprof.py`
- Autor: Robert Kern, URL: `https://github.com/rkern/line_profiler`
- Installation: `sudo pip install line_profiler`

Zeilenweises Profiling

- `cProfile` gibt Anzahl der Funktionsaufrufe und Laufzeit jeder Funktion an.
- Keine Auskunft darüber, welche Zeile wieviel Zeit benötigt.
- Lösung: `line_profiler` ermöglicht zeilenweises Profil.
- Skript zum Aufruf von `line_profiler`: `kernprof.py`
- Autor: Robert Kern, URL: `https://github.com/rkern/line_profiler`
- Installation: **`sudo pip install line_profiler`**

Aufruf

Auf der Kommandozeilen:

```
$ kernprof -l -v tiny_search_engine.py
```

`-l` Aktiviert den Line Profiler.

`-v` Ausgabe wird angezeigt.

Default Ausgabe wird in (Binär-)datei `<Scriptname>.py.lprof` geschrieben.

Line Profiler anwenden

- Um den `line_profiler` auf eine Funktion anzuwenden, müssen wir ihm das erst mitteilen
- Dazu "dekoriert" man die Funktion mit `@profile`. Beispiel:

```
@profile
def pseudo_data(l):
    v = []
    for i in range(l):
        v.append(random.randint(0,100))
    return v
```

Decorators

- Ein Decorator ist ein aufrufbares Python-Objekt, das auf eine Funktion, Methode oder Klassendefinition angewendet wird.
- Der Decorator wird angewendet, indem man `@decoratorname` vor die Definition schreibt.
- In diesem Fall wird also der Decorator `profile` auf zu profilierende Funktionen angewendet.
- Das so veränderte Skript lässt sich nicht mehr normal ausführen!

line_profiler Ausgabe

Profiler Ausgabe für die Funktion pseudo_data

```
$ kernprof -l -v tiny_search_engine.py -v 1000 -d 1000
```

```
Timer unit: 1e-06 s

Total time: 3.52772 s
File: tiny_search_engine.py
Function: pseudo_data at line 32

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    32                                           @profile
    33                                           def pseudo_data( l):
    34      1001          456      0.5      0.0      v = []
    35   1002001       316844      0.3      9.0      for i in range(l):
    36   1001000      3210057      3.2     91.0          v.append(random.randint(0,100))
    37      1001          368      0.4      0.0      return v
```

Übung 13