2020_topk

January 1, 2023

1 Algo - les k premiers éléments

Rechercher les k premiers éléments est un exercice classique d'algorithmie, souvent appelé *top-k*et qu'on peut résoudre à l'aide d'un algorithme de sélection. C'est très utile en machine learning pour retourner les 3, 4, ou 5 premiers résultats d'un modèle prédictif.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

[2]:
```
%matplotlib inline
```

1.1 Enoncé

On part d'une liste toute simple, une permutation aléatoire des premiers entiers. On connaît donc les k plus petits éléments qu'il faut obtenir.

[3]:
```
import numpy
perm = numpy.random.permutation(numpy.arange(20))
perm
```

[3]:
```
array([17, 11, 18, 15,  3, 19,  8, 14,  4,  2, 16,  5,  9,  7,  1,  6, 10,
        0, 13, 12])
```

1.1.1 Exercice 1 : on trie tout et on prend les k plus petit

C'est la méthode la plus simple mais pas la plus efficace.

[4]:
```
def topk_sortall(ensemble, k):
    # à vous
    # ...
    pass

topk_sortall(perm, 4)
```

1.1.2 Exercice 2 : version plus rapide au choix

A vois de choisir entre l'une et l'autre voire les deux si vous allez vite.
Idée 1
On garde un tableau de k éléments triés T, on parcourt tous les éléments du tableau initial. Si l'élément est plus petit que le plus grand des éléments dans le tableau T, on l'ajoute à ce tableau de façon à le garder trier. On continue jusqu'à la fin et le tableau T est l'ensemble des k plus petits éléments.
Idée 2
On découpe le tableau en $\left[\frac{n}{k}\right]$ tableaux de taille k qu'on trie. On fusionne ensuite ces $\left[\frac{n}{k}\right]$ pour garder ensuite les k plus petits éléments.

[5]:
```
def topk_idee1ou2(ensemble, k):
    pass
```

Vous avez aussi le droit de tricher pour une troisième idée Heap / Tas mais il faudra vous débrouiller tout seul.

1.1.3 Exercice 3 : algorithme de sélection

L'idée est assez simple : on prend un élément du tableau au hasard - le pivot - , on range à droite tous les éléments supérieurs, à gauche tous les éléments inférieurs. Si ce pivot se trouve à la position k alors on a trouvé les k plus petits éléments. Sinon, on sait où chercher.

[6]:
```

```

1.1.4 Exercice 4 : utiliser la fonction perf_counter pour comparer les vitessses de chaque solution

Et bien sûr, il faudra expliquer pourquoi l'algorithme le plus rapide est bien le plus rapide. Faire varier la taille du tableau initial.

[7]:
```
from time import perf_counter
```

[8]:
```

```

1.2 Réponses

1.2.1 Exercice 1 : on trie tout et on prend les k plus petit

Très simple mais le coût est en $O(n \ln n)$ puisqu'il faut trier tous les éléments.

[9]:
```
def topk_sortall(ensemble, k):
    ensemble = ensemble.copy()
    ensemble.sort()
    return ensemble[:k]

topk_sortall(perm, 4)
```

[9]:
```
array([0, 1, 2, 3])
```

Or trier tous les éléments n'est pas nécessaire. On n'a pas besoin de trier tous les éléments après la position k. Il est donc a priori envisageable de fairre mieux en ne faisant que les calculs nécessaires.

1.2.2 Exercice 2 : version plus rapide au choix

La première idée n'est pas une de celles proposées mais elle est très simple. Parmi les tris décrits sur cette interstices/tri, le tri par sélection est intéressant car il suffit d'effectuer les k premières itérations pour retrouver les k plus petits éléments aux k premières positions.

[10]:
```
def topk_tri_selection(ensemble, k):
    ensemble = ensemble.copy()
    for i in range(0, min(k, len(ensemble))):
        for j in range(i + 1, len(ensemble)):
            if ensemble[i] > ensemble[j]:
                ensemble[i], ensemble[j] = ensemble[j], ensemble[i]
    return ensemble[:k]

topk_tri_selection(perm, 4)
```

[10]:
```
array([0, 1, 2, 3])
```

Le coût de l'algorithme est en $O(kn)$.
idée 1

[11]:
```
def topk_insertion(ensemble, k):

    def position(sub, value):
        a, b = 0, len(sub) - 1
        m = (a + b) // 2
        while a < b:
            if value == sub[m]:
                return m
            if value < sub[m]:
                b = m - 1
            else:
                a = m + 1
            m = (a + b) // 2
        return m if sub[m] >= value else m + 1

    res = [ensemble[0]]
    for i in range(1, len(ensemble)):
        if ensemble[i] < res[-1]:
            # on insère selon un méthode dichotomique
            p = position(res, ensemble[i])
            res.insert(p, ensemble[i])
            if len(res) > k:
                res.pop()

    return res


topk_insertion(perm, 4)
```

[11]:
```
[0, 1, 2, 3]
```

Le coût de l'algorithme est dans le pire des cas $O(n * (\ln k + k))$:

- n car il y n itérations
- $\ln k$ car on recherche un élément de façon dichotomique dans un tableau trié

- k car on insère un élément dans un tableau en déplaçant tous les éléments plus grands

Dans les faits, c'est un peu plus compliqué car le fait qu'on ait trouvé un élément plus petit à la position i que le plus grand déjà trouvé n'est pas un événement fréquent si on suppose que le tableau est dans un ordre aléatoire. On pourrait même dire que la probabilité de trouver un élément faisant partie des k plus petits éléments est de plus en plus faible. On pourrait soit en rester là, soit se dire qu'on souhaite un algorithme plus rapide encore même dans le pire des cas qui serait pour cette version-ci un tableau trié dans le sens décroissant. Dans ce cas, on sait que le prochain élément fait toujours partie des k plus petits éléments connus jusqu'à présent.
On pourrait répondre à cette question si on suppose que les éléments sont indépendants et tirés selon la même loi aléatoire.
idée 2
L'idée est un peu plus compliquée.

[12]:
```
def topk_fusion(ensemble, k):
    n = len(ensemble) // (len(ensemble) // k)
    res = []
    last = 0
    while last < len(ensemble):
        res.append(last)
        last += n
    res.append(len(ensemble))

    subs = []
    for i in range(1, len(res)):
        sub = ensemble[res[i-1] : res[i]]
        sub.sort()
        subs.append(sub)

    indices = [0 for sub in subs]
    topk = []
    while len(topk) < k:
        arg = None
        for i, sub in enumerate(subs):
            if indices[i] < len(sub) and (arg is None or sub[indices[i]] <␣
↪subs[arg][indices[arg]]):
                arg = i
        topk.append(subs[arg][indices[arg]])
        indices[arg] += 1

    return topk


topk_fusion(perm, 4)
```

[12]: [0, 1, 2, 3]

1.2.3 Exercice 3 : algorithme de sélection

Première version par récurrence.

[13]:
```
def topk_select_recursive(ensemble, k):
    if len(ensemble) <= k:
        return ensemble
    p = ensemble[k]
```

```
    inf, sup = [], []
    for e in ensemble:
        if e > p:
            sup.append(e)
        else:
            inf.append(e)
    if len(inf) == k:
        return inf
    if len(sup) == 0:
        # potentiellement une boucle infinie, on change
        # le pivot de côté
        sup.append(p)
        del inf[inf.index(p)]
    if len(inf) >= k:
        return topk_select_recursive(inf, k)
    return inf + topk_select_recursive(sup, k - len(inf))


topk_select_recursive(perm, 4)
```

[13]: [3, 2, 1, 0]

Une seconde version sans récurrence.

[14]:
```
def topk_select(ensemble, k):
    ensemble = ensemble.copy()

    def loop(a, b, pivot):
        pivot_index = None
        while a < b:
            while a < len(ensemble) and ensemble[a] < pivot:
                a += 1
            while b >= 0 and ensemble[b] >= pivot:
                if ensemble[b] == pivot:
                    pivot_index = b
                b -= 1
            if a < b:
                ensemble[a], ensemble[b] = ensemble[b], ensemble[a]
        return min(a, len(ensemble) - 1), pivot_index


    a = 0
    b = len(ensemble) - 1
    m = k
    while True:
        pivot = ensemble[m]
        m, pi = loop(a, b, pivot)
        if m == k:
            return ensemble[:m]
        # Les cas particuliers interviennent lorsque pivot choisi est
        # identique au précédent.
        if m > k:
            if b == m - 1:
                ensemble[b], ensemble[pi] = ensemble[pi], ensemble[b]
```

```
                b -= 1
            else:
                b = m - 1
        else:
            if a == m:
                ensemble[a], ensemble[pi] = ensemble[pi], ensemble[a]
                a += 1
            else:
                a = m
        m = k


topk_select(perm, 4)
```

[14]: array([0, 1, 2, 3])

1.2.4 Exercice 4 : utiliser la fonction perf_counter pour comparer les vitessses de chaque solution

[15]:
```
from time import perf_counter

def measure_time(fct, perm, k, repeat=1000):
    t1 = perf_counter()
    for r in range(repeat):
        fct(perm, k)
    return perf_counter() - t1

measure_time(topk_sortall, perm, 4)
```

[15]: 0.0033986000000005845

[16]:
```
measure_time(topk_tri_selection, perm, 4)
```

[16]: 0.09221089999999954

[17]:
```
measure_time(topk_insertion, perm, 4)
```

[17]: 0.01921359999999961

[18]:
```
measure_time(topk_fusion, perm, 4)
```

[18]: 0.04859970000000047

[19]:
```
measure_time(topk_select_recursive, perm, 4)
```

[19]: 0.016174200000000027

[20]:
```
measure_time(topk_select, perm, 4)
```

[20]: 0.0986205

La première fonction est plus rapide surtout parce que le tri est implémenté par une fonction du langage particulièrement optimisée (en C). Il faudrait implémenter un tri en python pour pouvoir comparer.

[21]:
```
perm10000 = numpy.random.permutation(numpy.arange(10000))
```

[22]:
```
measure_time(topk_sortall, perm10000, 40, repeat=10)
```

[22]: 0.0050952000000004105

[23]:
```
measure_time(topk_tri_selection, perm10000, 40, repeat=10)
```

[23]: 1.6007581999999996

[24]:
```
measure_time(topk_insertion, perm10000, 40, repeat=10)
```

[24]: 0.10587389999999974

[25]:
```
measure_time(topk_fusion, perm10000, 40, repeat=10)
```

[25]: 0.1086203999999995

[26]:
```
measure_time(topk_select_recursive, perm10000, 40, repeat=10)
```

[26]: 0.03971849999999222

[27]:
```
measure_time(topk_select, perm10000, 40, repeat=10)
```

[27]: 0.09422410000001946

Il faut faire tourner la fonction un grand nombre de fois sur différentes tailles de n pour voir l'évolution.

[28]:
```
from tqdm import tqdm
from pandas import DataFrame

perm1000 = numpy.random.permutation(numpy.arange(1000))
res = []
for k in tqdm(range(100, 501, 50)):
    obs = dict(k=k, n=perm1000.shape[0])
    for fct in [topk_sortall, topk_tri_selection, topk_insertion,
                topk_fusion, topk_select_recursive, topk_select]:
        obs[fct.__name__] = measure_time(fct, perm1000, k, repeat=10)
    res.append(obs)

df = DataFrame(res)
df.head()
```

```
100%|¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ | 9/9 [00:09<00:00,  1.10s/it]
```

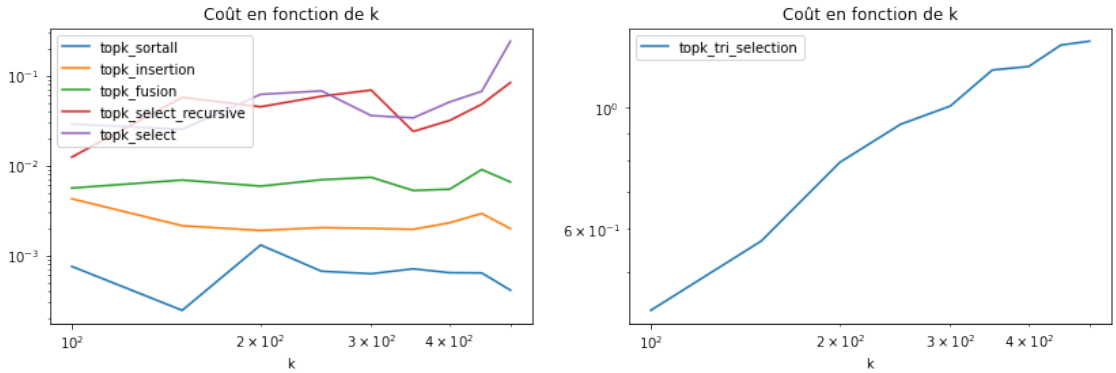
[28]:

	k	n	topk_sortall	topk_tri_selection	topk_insertion	topk_fusion	\
0	100	1000	0.000754	0.425342	0.004273	0.005617	
1	150	1000	0.000244	0.570124	0.002137	0.006894	
2	200	1000	0.001304	0.794225	0.001893	0.005902	
3	250	1000	0.000666	0.933013	0.002037	0.006944	
4	300	1000	0.000625	1.007520	0.001996	0.007394	

```
   topk_select_recursive  topk_select
```

```
0                0.012390     0.028960
1                0.057368     0.025287
2                0.045107     0.062032
3                0.059049     0.067764
4                0.069228     0.035983
```

[29]:
```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(1, 2, figsize=(14, 4))
df.set_index('k').drop(["n", "topk_tri_selection"], axis=1).plot(
    logy=True, logx=True, title="Coût en fonction de k", ax=ax[0])
df.set_index('k')[["topk_tri_selection"]].plot(
    logy=True, logx=True, title="Coût en fonction de k", ax=ax[1]);
```



Pas évident, le langage est trop haut niveau et le coût de l'interpréteur gomme les aspects algorithmique. Il faudrait augmenter la taille du tableau à trier ou affiner la mesure.

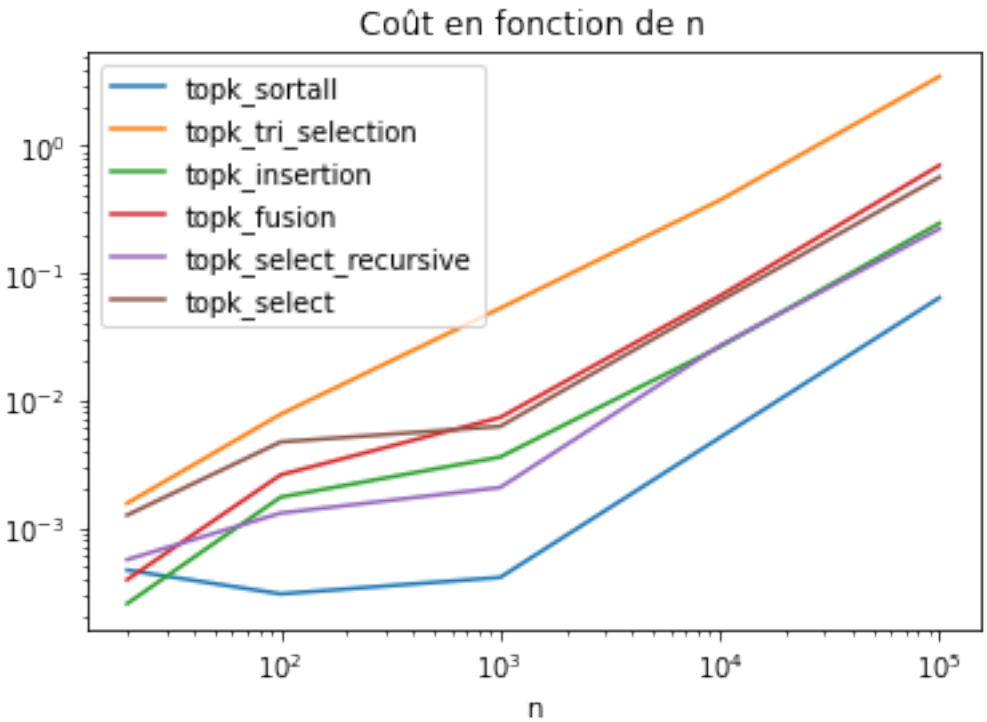
[30]:
```
res = []
k = 10
for n in tqdm([20, 100, 1000, 10000, 100000]):
    permn = numpy.random.permutation(numpy.arange(n))
    obs = dict(k=k, n=permn.shape[0])
    for fct in [topk_sortall, topk_tri_selection, topk_insertion,
                topk_fusion, topk_select_recursive, topk_select]:
        obs[fct.__name__] = measure_time(fct, permn, k, repeat=10)
    res.append(obs)

df = DataFrame(res)
df.head()
```

```
100%|¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿| 5/5 [00:05<00:00,  1.19s/it]
```

[30]:
```
    k       n  topk_sortall  topk_tri_selection  topk_insertion  topk_fusion  \
0  10      20      0.000464            0.001552        0.000253     0.000390
1  10     100      0.000301            0.007772        0.001735     0.002594
2  10    1000      0.000406            0.052845        0.003583     0.007324
3  10   10000      0.005089            0.372522        0.026215     0.066013
4  10  100000      0.063857            3.491449        0.245375     0.699281
```

```
   topk_select_recursive  topk_select
0               0.000560     0.001252
1               0.001299     0.004708
2               0.002064     0.006233
3               0.026855     0.060124
4               0.221890     0.565102
```

[31]:
```
df.set_index('n').drop(["k"], axis=1).plot(
    logy=True, logx=True, title="Coût en fonction de n");
```



Les coûts paraissent linéaires en n. Il faudrait cependant augmenter la taille encore car on sait que l'un d'entre eux ne l'est pas. Le coût de l'interpréteur python est là-encore non négligeable. Il faudrait prendre des langages plus rapide ou plus bas niveau ou augmenter les tailles pour voir quelque chose d'intéressant.

[32]:
```
```