TD #6: Random projections

Large-scale Mathematical Programming

Leo Liberti, CNRS LIX Ecole Polytechnique
liberti@lix.polytechnique.fr

INF580

Outline

Verifying the JLL

- The JLL depends on two parameters:
  1. multiplicative approximation accuracy ε
  2. multiplicative factor C in $k = O(\varepsilon^{-2} \ln n)$

  where $n =$ number of pts in $\mathbb{R}^m$ to project to $\mathbb{R}^k$
- Consider $m \times n$ point matrix X
  sampled from $\mathrm{U}(0,1)$ *and* $\mathrm{N}(0,1)$
- Sample random projector (RP) T in $\mathrm{N}(0, 1/\sqrt{k})$
- Verify projection err for cols of TX w.r.t. cols of X

Projection errors

Given Euclidean Distance Matrices (EDM) D of X, D' of TX, compute the following error measures:

1. $J = \{\max(0, |\frac{D'_{ij}}{D_{ij}} - 1| - \varepsilon) \mid i, j \leq n\}$
2. $J_{\text{card}} = \sum_{\substack{e \in J \\ e > 0}} 1$
3. $J_{\text{avg}} = \frac{1}{n^2} \sum_{e \in J} e$
4. $J_{\max} = \max(J)$
5. $\text{mde} = \frac{1}{n^2} \sum_{i,j \leq n} |D_{ij} - D'_{ij}|$
6. $\text{lde} = \max_{i,j \leq n} |D_{ij} - D'_{ij}|$

Necessary tasks

- sample random matrices from $\mathrm{U}(0,1)$ and $\mathrm{N}(0,1)$

  you can also try changing distribution parameters
- fast computation of large distance matrices
- fast dot product of large matrices

Sampling random matrices

Python: `import numpy as np`

- $m \times n$ matrix sampled componentwise from $\mathsf{U}(0,1)$
  `np.random.rand(m,n)`
- $m \times n$ matrix sampled componentwise from $\mathsf{N}(0,1)$
  `np.random.normal([0],[[1]],(m,n))`

Fast computation of distance matrices

Python: `from scipy.spatial.distance import pdist`

- `X` is a `numpy` array with n cols in $\mathbb{R}^m$
- `D = pdist(X.T)`
  `pdist` considers *row vectors*, so we need $X^\top$
  `pdist` returns upper triangular part of D
  encoded as $(n(n-1)/2)$-vector

Fast matrix product

Given matrices T $(k \times m)$ and X $(m \times n)$:

- dense matrices:

```
import numpy as np
TX = np.dot(T,X)
```

- sparse matrices in CSR format:

```
import scipy.sparse
TX = scipy.sparse.csr_matrix.dot(T,A)
```

- if using Python 2:

```
from blis.py import gemm
TX = gemm(T,X)
```

Generating the points

```
import sys
import math
import numpy as np
import scipy.sparse

# uniform dense
X = np.random.rand(m,n)

# normal dense
X = np.random.normal([0],[[1]],(m,n))

# normal sparse (density s, CSR format)
X = scipy.sparse.random(m,n,density=s,format='csr',
    data_rvs=np.random.randn)
```

Main loop

```
D = pdist(X.T)
nD = len(D)
for eps in [0.05, 0.1, 0.15, 0.2]:
    print(" --------------------------")
    for C in [0.5, 1.0, 1.5, 2.0]:
        k = int(round(C*(1/eps**2)*math.log(n)))
        T = (1/sqrt(k))*normalmatrix(k,m)
        TX = np.dot(T,X)
        TD = pdist(TX.T)
```

Computing the errors

```
jllerr = [max(0, abs(TD[i]/D[i]-1)-eps) for i in range(nD)]
jllerr = [jle for jle in jllerr if jle > myZero]
jllerr = len(jllerr)
avgjllerr = sum(abs(TD[i]/D[i]-1) for i in range(nD)) / nD
maxjllerr = max(abs(TD[i]/D[i]-1) for i in range(nD))
mde = sum(abs(D[i] - TD[i]) for i in range(nD)) / nD
lde = max(abs(D[i] - TD[i]) for i in range(nD))
```

Results on 2000×1000 matrix from $U(0, 1)$

ε	C	k	jllerr	avgjll	maxjll	mde	lde
0.05	0.5	1382	3919	0.015	0.09	0.273	1.684
0.05	1.0	2763	118	0.011	0.063	0.197	1.151
0.05	1.5	4145	4	0.009	0.053	0.16	0.976
0.05	2.0	5526	0	0.008	0.044	0.14	0.817
0.1	0.5	345	3504	0.029	0.194	0.536	3.615
0.1	1.0	691	101	0.021	0.13	0.391	2.372
0.1	1.5	1036	2	0.017	0.107	0.318	1.926
0.1	2.0	1382	0	0.015	0.085	0.273	1.541
0.15	0.5	154	3942	0.045	0.272	0.823	4.983
0.15	1.0	307	108	0.032	0.193	0.59	3.454
0.15	1.5	461	2	0.026	0.163	0.481	2.957
0.15	2.0	614	0	0.023	0.134	0.412	2.457
0.2	0.5	86	4675	0.062	0.373	1.13	6.88
0.2	1.0	173	58	0.043	0.271	0.776	4.969
0.2	1.5	259	11	0.037	0.252	0.668	4.656
0.2	2.0	345	0	0.03	0.18	0.549	3.281

Results on 2000×1000 matrix from $N(0, 1)$

ε	C	k	jllerr	avgjll	maxjll	mde	lde
0.05	0.5	1382	4170	0.015	0.094	0.952	5.961
0.05	1.0	2763	98	0.011	0.064	0.686	4.149
0.05	1.5	4145	0	0.009	0.05	0.554	3.127
0.05	2.0	5526	0	0.008	0.045	0.485	2.883
0.1	0.5	345	3560	0.03	0.209	1.877	13.173
0.1	1.0	691	145	0.022	0.139	1.389	8.717
0.1	1.5	1036	2	0.018	0.11	1.127	6.824
0.1	2.0	1382	0	0.015	0.094	0.975	5.971
0.15	0.5	154	4589	0.046	0.282	2.891	17.811
0.15	1.0	307	120	0.032	0.212	2.053	13.294
0.15	1.5	461	10	0.027	0.169	1.68	10.824
0.15	2.0	614	0	0.022	0.13	1.42	8.41
0.2	0.5	86	4498	0.061	0.402	3.878	25.188
0.2	1.0	173	74	0.042	0.244	2.681	15.296
0.2	1.5	259	1	0.035	0.202	2.205	13.089
0.2	2.0	345	0	0.03	0.174	1.911	10.999

Comparative results

JLL tests on 2000 x 1000 random data point matrix

			Unif[0,1)					Normal(0,1)					Differences				
eps	C	k	jllerr	avgerr	maxerr	mde	lde	jllerr	avgerr	maxerr	mde	lde					
0.05	0.5	1382	4170	15	94	952	5961	3919	15	0.09	273	1684	251	0	93.91	679	4277
0.05	1.0	2763	98	11	64	686	4149	118	11	63	197	1151	-20	0	1	489	2998
0.05	1.5	4145	0	9	0.05	554	3127	4	9	53	0.16	976	-4	0	-52.95	553.84	2151
0.05	2.0	5526	0	8	45	485	2883	0	8	44	0.14	817	0	0	1	484.86	2066
0.1	0.5	345	3560	0.03	209	1877	13173	3504	29	194	536	3615	56	-29	15	1341	9558
0.1	1.0	691	145	22	139	1389	8717	101	21	0.13	391	2372	44	1	138.87	998	6345
0.1	1.5	1036	2	18	0.11	1127	6824	2	17	107	318	1926	0	1	-106.89	809	4898
0.1	2.0	1382	0	15	94	975	5971	0	15	85	273	1541	0	0	9	702	4430
0.15	0.5	154	4589	46	282	2891	17811	3942	45	272	823	4983	647	1	10	2068	12828
0.15	1.0	307	120	32	212	2053	13294	108	32	193	0.59	3454	12	0	19	2052.41	9840
0.15	1.5	461	10	27	169	1.68	10824	2	26	163	481	2957	8	1	6	-479.32	7867
0.15	2.0	614	0	22	0.13	1.42	8.41	0	23	134	412	2457	0	-1	-133.87	-410.58	-2448.59
0.2	0.5	86	4498	61	402	3878	25188	4675	62	373	1.13	6.88	-177	-1	29	3876.87	25181.12
0.2	1.0	173	74	42	244	2681	15296	58	43	271	776	4969	16	-1	-27	1905	10327
0.2	1.5	259	1	35	202	2205	13089	11	37	252	668	4656	-10	-2	-50	1537	8433
0.2	2.0	345	0	0.03	174	1911	10999	0	0.03	0.18	549	3281	0	0	173.82	1362	7718

The Achlioptas sparse projector

- Let T be sampled componentwise from the distribution:

$$T_{ij} \sim \begin{cases} -1 & \text{with prob. } p/2 \\ 0 & \text{with prob. } 1-p \\ 1 & \text{with prob. } p/2 \end{cases}$$

- For $p = 1/3$ we get [Achlioptas 2003]'s RP
- For $p = 1/\sqrt{m}$ we get [Li, Hastie, Church 2006]'s RP
- **Scaling:** for density p, pre-multiply by $1/\sqrt{pk}$
- Use unscaled $T \in \{-1, 0, 1\}^{km}$ to compute TX, then scale

  reduces time for floating point computations

Tasks

- Verify the JLL with Achlioptas' projectors
- Consider the $k \times m$ sparse RP S with density p

$$S \sim \frac{1}{\sqrt{pk}} (\mathrm{N}(0,1) \text{ with prob. } p)$$

- Verify the JLL with S

Outline

Images

- Read all image files in a given directory
- Scale them to identical size and color depth
- Transform them into set X of vectors in $\mathbb{R}^m$
- Cluster X using K-means (with given K)
- Randomly project X to $Y \subset \mathbb{R}^k$ where $k = O(\ln |X|)$
- Cluster Y using K-means
- Compare clusterings and timings for different image folders
- **Task:** *simply put together the code from the various parts and use it*

Structure of the python code

1. Read all files in a given dir: `glob.glob`
2. Read, scale, convert images: `PIL.Image`
3. K-means: `sklearn.cluster.KMeans`
4. CPU time: `time.time`
5. Compare clusterings: `sklearn.metrics.cluster.adjusted_mutual_info_score`

Imports

```
import sys
import os
import time
import math
from math import sqrt
import numpy as np
from PIL import Image
import glob
from sklearn.cluster import KMeans
from sklearn.metrics.cluster import adjusted_mutual_info_score
```

Global parameters

```
myZero = 1e-10
image_exts = [".jpg",".gif",".png"]
thumbnailsize = (100,100)
thumbnaildepth = 3
jlleps = 0.15
jllC = 2.0
```

Functions

```
# round and output a number as part of a string
def outstr(x,d):
    return str(round(x,d))

# generate a componentwise Normal(0,1) matrix
def normalmatrix(m, n):
    return np.random.normal([0],[[1]],(m,n))

# generate a componentwise Uniform(0,1) matrix
def uniformmatrix(m, n):
    return np.random.rand(m,n)
```

Functions

```
def outclustering(clust, filenames):
    nclust = len(set(list(clust.labels_)))
    for c in range(nclust):
        print("  " + str(c+1) + ":", end=' ')
        for j in range(n):
            if clust.labels_[j] == c:
                print(filenames[j], end=' ')
        print()

def nonemptyclust(clust):
    nclust = len(set(list(clust.labels_)))
    clustering = {}
    for c in range(nclust):
        cluster = [j for j in range(n) if clust.labels_[j] == c]
        if len(cluster) > 0:
            clustering[c] = cluster
    return clustering
```

Read command line

```
if len(sys.argv) < 3:
    print("syntax is" + sys.argv[0] + "dir nclust")
    print("  dir contains image files")
    print("  nclust is number of clusters")
    sys.exit(1)

dir = sys.argv[1]
nclust = int(sys.argv[2])
if nclust < 2:
    sys.exit('nclust must be at least 2')

if len(sys.argv) >= 4:
    m = int(sys.argv[2])
    n = int(sys.argv[3])
    thumbnailsize = (m,n)

m = thumbnailsize[0]*thumbnailsize[1]*thumbnaildepth
```

Read files into vectors

```
filenames = []
X = []
n = 0
print("reading " + dir + " ...", end=' ')
sys.stdout.flush()
t0 = time.time()
for ext in image_exts:
    for filename in glob.glob(dir + '/*' + ext):
        im = Image.open(filename)
        im = im.resize(thumbnailsize)
        im = im.convert("RGB")
        filenames.append(os.path.basename(filename))
        imvect = np.reshape(np.array(im), (m))
        X.append(imvect)
        n += 1
X = np.array(X)
t1 = time.time()
print("took " + outstr(t1-t0,2) + "s")
```

K-means

```
## cluster the data matrix
print(str(nclust) + "-means clustering ...", end=' ')
sys.stdout.flush()
t2 = time.time()
clust = KMeans(n_clusters=nclust).fit(X)
t3 = time.time()
#outclustering(clust, filenames)
print("took " + outstr(t3-t2,2) + "s")
```

Random projection

```
## projecting the data matrix
print("projecting data matrix ...", end=' ')
sys.stdout.flush()
t4 = time.time()
k = int(round(jllC*(1/(jlleps**2))*math.log(n)))
T = (1/sqrt(k))*normalmatrix(m,k)
XT = np.dot(X,T)
t5 = time.time()
print("took " + outstr(t5-t4,2) + "s")
print("projected from " + str(m) + " to " + str(k) + " dims")
```

K-means on projected data

```
## projected k-means clustering
print(str(nclust) + "-means proj. clustering ...", end=’
sys.stdout.flush()
projclust = KMeans(n_clusters=nclust).fit(XT)
t6 = time.time()
#outclustering(projclust, filenames)
print("took " + outstr(t6-t5,2) + "s")
print("clust took "+outstr(t3-t2,2)+"s;", end=’ ’)
print("proj+clust took "+outstr(t6-t4,2)+"s")
```

Clustering similarity

```
## evaluate clustering similarity
print("used " + multmethod + " for matrix dot product")
q = adjusted_mutual_info_score(clust.labels_, projclust.labels_)
print("adj mutual info = " + outstr(q,3) + " (0=differ, 1=equal)")
clustering = nonemptyclust(clust)
print(str(n) + " images clustered into", end=' ')
print(str(len(clustering.keys())) + " non-empty clusters")
```

Output: 13 images and $K = 3$

```
reading ~/gif/art/ ... took 2.07s
3-means clustering ... took 0.12s
projecting data matrix ... took 0.31s
projected from 30000 to 228 dims
3-means proj. clustering ... took 0.01s
clust took 0.12s; proj+clust took 0.32s
used numpy.dot for matrix dot product
adj mutual info = 1.0 (0=different, 1=equal)
13 images clustered into 3 non-empty clusters
```

Output: 12 images and $K = 4$

```
reading ~/gif/places/ ... took 1.34s
4-means clustering ... took 0.11s
projecting data matrix ... took 0.29s
projected from 30000 to 213 dims
4-means proj. clustering ... took 0.01s
clust took 0.11s; proj+clust took 0.31s
used numpy.dot for matrix dot product
adj mutual info = 1.0 (0=different, 1=equal)
12 images clustered into 4 non-empty clusters
```

Output: 88 images and $K = 4$

```
reading ~/gif/things/ ... took 5.61s
4-means clustering ... took 0.88s
projecting data matrix ... took 0.57s
projected from 30000 to 396 dims
4-means proj. clustering ... took 0.02s
clust took 0.88s; proj+clust took 0.59s
used numpy.dot for matrix dot product
adj mutual info = 0.847 (0=different, 1=equal)
88 images clustered into 4 non-empty clusters
```

Output: 244 images and $K = 3$

```
reading ~/gif/foods/ ... took 29.47s
5-means clustering ... took 3.33s
projecting data matrix ... took 0.94s
projected from 30000 to 487 dims
5-means proj. clustering ... took 0.1s
clust took 3.33s; proj+clust took 1.04s
used numpy.dot for matrix dot product
adj mutual info = 0.719 (0=different, 1=equal)
240 images clustered into 3 non-empty clusters
```

Output: 395 images and $K = 10$

```
reading ~/gif/people/ ... took 25.29s
10-means clustering ... took 10.47s
projecting data matrix ... took 1.07s
projected from 30000 to 531 dims
10-means proj. clustering ... took 0.27s
clust took 10.47s; proj+clust took 1.34s
used numpy.dot for matrix dot product
adj mutual info = 0.519 (0=different, 1=equal)
395 images clustered into 10 non-empty clusters
```

Output: 395 images and $K = 3$

```
reading ~/gif/people/ ... took 25.58s
3-means clustering ... took 5.98s
projecting data matrix ... took 1.07s
projected from 30000 to 531 dims
3-means proj. clustering ... took 0.13s
clust took 5.98s; proj+clust took 1.21s
used numpy.dot for matrix dot product
adj mutual info = 0.729 (0=different, 1=equal)
395 images clustered into 3 non-empty clusters
```

Outline

The *diet problem*

Given:

- a set F of n possible foods in the diet, with unit costs $c_j \sim \max(0.1, 1 + 0.1\,\mathrm{N}(0,1))$
  let c be the food cost vector
- a set N of m nutrients the diet must provide, in quantity at least b_i
  let b be the required nutrient quantity vector
- values $a_{ij} \sim \mathrm{U}(0.05, 2.5)$ such that food j contains a_{ij} units of nutrient i
  let A be the nutrient-food occurrence matrix
  make sure A has density 0.1
- find the diet of least cost

Tasks

- Formulate the diet problem
- Write an AMPL or Python script in order to generate random *feasible* instances A, b, c of given sizes m, n of the diet problem
  [Hint: generate b last]
- Write a Python script to read these instances
- Write Python code to apply a RP to the diet problem LP
- Solve both original and projected problem using `amplpy`
- Compare results (approximation quality, feasibility, CPU time) for instances of various sizes

Solution retrieval is tricky

- With AMPL: `sstatus` suffix `if x[j].sstatus == "bas" ...`
- With `amplpy`: `xstat[j] = diet.getVariable("x")[j+1].sstatus()`
- Basis elements come from simplex tableau
  they may refer to slack or Phase I variables
  $\Rightarrow$ must look for them in constraint structures too
  `cstat[i] = diet.getConstraint("nutrients")[i+1].sstatus()`
- $\Rightarrow$ Since problem constraints are $Ax \geq b$ you must project $\bar{A} = (A|I_m)$
  as RPs are applied to LPs in standard form (with linear equations)
  var (resp. constr) basic elts indexed in $\{1, \ldots, n\}$ *(resp.* $\{1, \ldots, m\}$*)*
  but i-th basic constr. index refers to $(n+i)$-th col in $\bar{A}$
- When you retrieve $x' = (A_H^\top A_H)^{-1} A_H^\top b$ (see lectures), x'_j may correspond to a basic element from variables or constraints
  $\Rightarrow$ zero padding will need to fill in "missing indices"
  retrieved x will be in $\mathbb{R}^{n+m}$
  [Hint: encode x' in a dictionary: `xd[j]` *= value of x'_j]*

CPLEX on original 4000×5000 instance

```
CPLEX 12.8.0.0: display=1
Parallel mode: deterministic, using up to 4 threads for concurrent optimization.
Linear dependency checker was stopped due to maximum work limit.
No LP presolve or aggregator reductions.
Elapsed time = 63.74 sec. (61801.03 ticks, 1 iterations)
Dual simplex solved model.
CPLEX 12.8.0.0: optimal solution; objective 793.258801
2690 dual simplex iterations (0 in phase I)
cost = 793.259
real 2m56.951s, user 4m8.926s, sys 0m5.516s
```

With random projection code ($\varepsilon = 0.08,\ C = 1$)

```
reading data   data read in 6.90s
projecting from 4000 to 1423 constraints   projection took 1.75s
writing projected problem to projdiet.dat   wrote instance in 21.38s
passing data to AMPL using file   passed data to AMPL in 3.82s
solving projected problem   solved projected problem in 90.17s
CPLEX 12.8.0.0: baropt
bardisplay=1
Linear dependency checker was stopped due to maximum work limit.
No LP presolve or aggregator reductions.
Parallel mode: using up to 4 threads for barrier.
CPLEX 12.8.0.0: optimal solution; objective 793.2632
10 barrier iterations
9 push, 792 exchange dual crossover iterations
solution retrieval   retrieval took 0.77s
optimal objective function value = 793.2632
||(A|I_m)xretr - b||_2 / m = 1.5309781918953325e-14
||min(xretr,0)||_1 / (n+m) = 2.4976317310650604e-15
CPU times: read=6.90,proj=1.75,out=21.38,solve=90.17,retr=0.77,tot=120.98
real 2m7.021s, user 4m13.557s, sys 0m4.257s
```

A last warning

If you obtained error-free results (like me in last slide), you probably biased the random instance in the generation phase!

Find what the bias is, repair the random generation code, and repeat the tests: what results do you obtain?