Sparse Spectral Hashing

Group #10

Problem statement

Fast finding of similar data points to a given query from a large scale database using Sparse Spectral Hashing

Methodology

Assume that we have a collection of N d-dimensional datapoints in Euclidean space $\{(\boldsymbol{x}_i) \in \mathbb{R}^d : i = 1, 2, \ldots, N\}$, where $\boldsymbol{x}_i$ represents the feature vector for the i^{th} datapoint and d represents the dimension of the features from the training data. Θ is an efficient indexing function to map each x_i from d-dimensional Euclidean space to m-dimensional Hamming space y_i. We define Θ as follows :

$$\Theta : \boldsymbol{x}_i \in \mathbb{R}^d \rightarrow \boldsymbol{y}_i \in \{-1, 1\}^m$$

Feature Extraction

```
# Now we can do feature extraction using `forward1` as before. Notice that the last convolution layer uses 2048 channe

# We now add the classes we want to classify on. Here I added football and lion class.

# In[19]:

#extraction of features of our classes
from PIL import Image
import glob

out=[]
data_pos = np.array([get_image('football/'+img) for img in os.listdir('football/')])
data_neg = np.array([get_image('lion/'+img) for img in os.listdir('lion/')])
# data_pos1 = np.array([get_image('piano/'+img) for img in os.listdir('piano/')])
data_neg1 = np.array([get_image('guitar/'+img) for img in os.listdir('guitar/')])
datas = np.array([get_image('oxford/'+img) for img in os.listdir('oxford/')])

X = np.append(data_pos,data_neg,axis=0)
Y = np.append(data_neg1,datas,axis=0)
Z = np.append(X,Y,axis=0)

print Z, Z.shape

import pandas

for img in Z:
    mod3.forward(Batch([mx.nd.array(img)]))
    out.append(mod3.get_outputs()[0].asnumpy())
out = np.array(out)

print out,out.shape
```

Sparse PCA

- We aim to minimize $\sum_{ij} W_{ij}\|y_i - y_j\|^2$, which is the average Hamming distance between similar neighbours, with $y_i \in \{-1,1\}^k$, $\sum_i y_i = 0$ and $\frac{1}{n}\sum_i y_i y_i^T = I$

  where W is the affinity matrix, given by $W(i,j) = \exp(-\|x_i - x_j\|^2/\epsilon^2)$
- However, this is an NP-complete problem. This can be relaxed by removing the $y_i \in \{-1,1\}^k$ constraint and introducing a Laplacian matrix L, to turn this into a graph-partitioning problem. The solution will be the m eigenvectors with the minimum eigenvalues. Here, L = D - W, where D is a diagonal matrix, given by D(i,i) = $\sum_j W(i,j)$

- But, in PCA, all data coordinates take part in the linear combination. Thus, the Principal Components (PCs) lack in sparseness

Sparse PCA

- Hence, a sparse factor is introduced into the Spectral Hashing. The PCs are obtained from a linear combination of only certain, and not all, features. If p is the sparse loadings of the Laplacian L, then we minimize this : $p^T L p + \rho \, \mathrm{Card}^2(p)$ where Card(p) is the cardinality of p and ρ is the factor that controls the penalty
- However, this also turns out to be an NP-hard problem and requires a convex relaxation. Therefore, now, we minimize this : $\mathrm{trace}(LP) + \rho \, 1^T|P|1$, where $P = pp^T$
- This is now a semi-definite problem and can be solved iteratively

Sparse PCA

- Given the covariance matrix $\boldsymbol{\Sigma}$ of $\boldsymbol{X}$ ($\boldsymbol{\Sigma} \in \mathbb{R}^{d \times d}$), a d-dimensional sparse PC p can be obtained according to the convex relaxation
- Then the covariance matrix R is updated as follows :

$$\boldsymbol{\Sigma} := \boldsymbol{\Sigma} - (\boldsymbol{p}^T \boldsymbol{\Sigma} \boldsymbol{p}) \boldsymbol{p} \boldsymbol{p}^T$$

- This update of the covariance matrix is done over m iterations and convex relaxation is implemented to each updated $\boldsymbol{\Sigma}$.
- This way, we can get m sparse PCs and construct a matrix $\boldsymbol{M} \in \mathbb{R}^{d \times m}$ whose columns are the m sparse PCs
- The matrix **B** can then be obtained by matrix multiplication : **B = XM**

Sparse PCA

We denote the j^{th} **(j = 1, . . . , m)** column of matrix **B** as $\mathbf{B}_{(:,j)}$. Obviously, the cardinality of **B (:,j)** is **N**. For each $\mathbf{B}_{(:,j)}$, a function δ_j^k is defined as follows :

$$\delta_j^k = 1 - e^{-\frac{\varepsilon^2}{2}\left|\frac{k\pi}{B_{(:j)}^{max} - B_{(:j)}^{min}}\right|^2}$$

where k = 1,2,3,..m, $\mathbf{B}_{(:j)}^{\mathbf{max}}$ **and** $\mathbf{B}_{(:j)}^{\mathbf{min}}$ are respectively the maximum and minimum values in $\mathbf{B}_{(:j)}$ and epsilon is a constant.

Now, we rank these δ_j^k which is a matrix of m rows and m columns and choose the first m minimum ones

Representation of matrix B


```
31,2032,2033,2034,2035,2036,2037,2038,2039,2040,2041,2042,2043,2044,2045,2046,2047
0,0.459373235703,0.0,0.0319174937904,0.98147636652,0.0213114507496,0.0436733104289,0.0,0.0979871451855,0.14622206986
,0.0592861808836,0.0,0.709365189075,2.17543840408,0.294322878122,0.0592923797667,0.397848069668,1.34622657299,
0.240175962448,0.42408451438,0.0941695794463,0.21141923964,0.307677149773,0.0617188215256,0.0517627261579,0.
00328892096877,0.251478075981,0.0581732168794,0.306579053402,0.230239585042,0.0154754407704,0.0738295540214,
0.457439422607,0.249865055084,1.61377894878,0.0689147114754,0.00896345172077,1.09850513935,
0.132511541247,2.50674939156,0.336754918098,0.112496085465,0.719785451889,0.0,0.796805858612,0.15518181026,0.
0270097050816,0.0,1.6384125948,0.111411601305,1.25970840454,1.10076212883,0.318130552769,0.0,1.62704241276,0.
05647367239,0.323065668344,0.121424749494,0.0267780218273,0.177465200424,0.0991647690535,0.726827561855,0.0,
0.120607659221,0.2280575037,0.844089627266,0.297158360481,0.0993118733168,0.149878501892,
0.131939589977,3.13087105751,0.169664978981,1.09036016464,0.0422294028103,0.000761662668083,0.00624576630071,
0.413043856621,0.00223835231736,0.0025866930373,0.745532214642,0.0,0.435919344425,0.11852966994,0.113475143909,
0.56294631958,0.0,0.0338043384254,0.726018488407,0.189349040389,0.07091666013,0.0730329230428,0.0478869043291,
0.10820723325,0.00269332830794,0.484602510929,0.08186429739,0.000907566165552,2.31092166901,0.157231360674,0.
0078507354483,0.396248102188,0.0503837130964,0.0650722235441,0.215908870101,0.0766419991851,
0.263989567757,2.6900015655e-06,0.247340291739,0.144402503967,0.0977042466402,0.0,0.477165848017,0.0808675289154,
0.197517588735,0.0132228862494,0.267919957638,0.0,0.411760866642,0.93805450201,1.24116110802,1.43677532673,
0.787833213806,0.475378930569,0.0307698920369,0.187923997641,0.828530430794,0.00069937290391,0.139247894287,0.
0849484801292,0.250835001469,0.0925378277898,0.00848665647209,0.680232226849,0.819449901581,0.660922050476,0.
0865714550018,0.309376657009,0.764567375183,0.79349809885,0.170466199517,0.143978402019,0.986211538315,0.
0677150040865,0.746104896069,1.33564150333,0.0,0.00123581069056,0.192934125662,0.179164782166,0.024566071108,0.
00700015202165,0.0141955735162,0.0975018143654,0.327936738729,0.629732787609,0.136283069849,0.419862031937,0.
0303671509027,0.127578660846,0.16055034101,0.200133740902,0.0302084572613,0.297413766384,0.784494757652,
0.189086183906,0.261724591255,0.0,0.000832237419672,0.0979707092047,0.0104860756546,0.0163165796548,0.041079249233,
0.44390887022,0.0,0.17044788599,0.320688992739,0.0,0.12068542093,0.146356076002,0.121520623565,0.683639883995,
0.335662096739,0.236358299851,0.0479142330587,0.100420095026,0.393238902092,0.0301508158445,0.184232160449,0.
00127655314282,0.0,0.241166561842,0.240616515279,0.289678752422,0.023678638041,0.390825688839,0.332721114159,0.0,0.
0124141313136,0.101137526333,0.273095607758,0.0470687709749,0.0764993950725,0.122343569994,1.061891675,0.43948969245
,0.062342017889,0.269702643156,0.942800521851,0.0235772915184,0.027844119817,0.596822559834,0.735217154026,
0.393918484449,0.0,0.00224904017523,0.0472918599844,0.0498501770198,0.391479551792,0.103455111384,0.396602958441,
0.249739214778,0.0532015226781,0.0784767344594,0.0425767302513,0.195503875613,1.68040013313,0.462272942066,
0.132402330637,0.0567006506026,0.0326658673584,0.0833854079247,0.00275893346407,0.00652714679018,0.98872834444,0.
0142365228385,0.0566404387355,0.10571013391,0.0307460222393,0.213469028473,0.480280160904,0.809329748154,0.
0569092594087,0.00187169772107,0.513989746571,0.719115793705,0.41261857748,0.192756742239,0.150160759687,
0.344066739082,0.0326248183846,0.571996748447,0.468334019184,0.0427043475211,0.0216185897589,0.186305016279,
0.322673290968,0.673107624054,0.0130620477721,0.462387472391,0.512139499187,0.203257888556,0.0,0.298033028841,0.
0752599462867,0.018614590168,0.267645061016,0.781121969223,0.118180640042,1.75687217712,0.0261282324791,
0.314261049032,0.118769273162,0.0415642783046,0.39627340436,0.400084286928,0.236655518413,1.63355302811,
```

Sparse PCA

Given the **i**th image $\mathbf{x}_i$ in training set, let its corresponding binary code be $\boldsymbol{y}_i \in \{-1, 1\}^m$ and the value after mapping be $\boldsymbol{z}_i \in (-1, 1)^m$. The **j**th binary value **z(i, j)** in $\mathbf{z}_i$ can be calculated as :

$$\boldsymbol{z}(i,j) = \Theta\left(\delta_j^{min}, \boldsymbol{B}(i,t)\right) = \sin\left(\frac{\pi}{2} + \frac{k\pi}{\boldsymbol{B}_{(:,t)}^{max} - \boldsymbol{B}_{(:,t)}^{min}} \boldsymbol{B}(i,t)\right)$$

After obtaining **z(i, j)**, we take a threshold to generate the binary code **y(i, j)** in the following steps, through a boosting algorithm.

Computing Z(i,j)...

```python
def hamming_z(x):
    x = np.array(x)
    row,col = x.shape
    epsilon = 1e-1
    delta_kj = np.zeros((col,col))
    col_min = []
    col_max = []
    lister = []
    for j in range(col):
        e = x[:][j].min()
        f = x[:][j].max()
        col_min.append(e)
        col_max.append(f)
        for k in range(col):
            delta_kj[j][k] = ( 1 - 2.71**( epsilon*epsilon*0.5*abs(( (k * 3.14)*1.0 / (e - f) )*( (k * 3.14)*1.0 / (e - f) ))))
            lister.append(delta_kj[j][k])
    lister.sort()
    # print delta_kj
    col_min = np.array(col_min)
    col_max = np.array(col_max)
    lol = lister[:col]
    indexes = {}
    # print lol
    for l in range((col)):
        for i in range(col):
            for j in range(col):
                if delta_kj[i][j] == lol[l]:
                    indexes[l] = [i,j]
                    break
    # print indexes
    z = np.zeros((row,col))
    y = np.zeros((row,col))
    for u in range(row):
        for v in range(col):
            z[u][v] = math.sin(3.14/2 + ((indexes[v][1]*11*x[u][indexes[v][0]])*1.0*3.14/(col_max[indexes[v][0]] - col_min[indexes[v][0]])))
            if z[u][v] <= 0:
                y[u][v] = 1
            else:
                y[u][v] = -1
    return z,y
```

Binary coding by boosting

- Real world data does not always have uniform distribution. Thus, a data-aware and adaptive threshold has to be learned from the dataset. This is done through AdaBoost.
- Each column of the Z matrix is considered as a weak learner, to transform an encoding problem to a classifier problem
- A threshold T_j for each column is to be learned, after which binarization is done as follows :

$$\mathbf{y}(i,j) = \begin{cases} +1, \textbf{if } \mathbf{z}(i,j) \leqslant T_j \\ -1, \text{otherwise} \end{cases}$$

- We would also need to compare pairs of images and check if they are similar or not. This comparison will be done column-wise for all pairs of images. For example, f(u,v) = y(u,j) * y(v,j), where the u^{th} and v^{th} images are compared by their j^{th} columns. If they are similar, f(u,v) = 1. Else, f(u,v) = -1

Learning thresholds

- Obtain N^2 image pairs from the dataset of N images. For each of these pairs, assign a label f (= 1 or -1) depending on whether the images in the pair are similar or not
- We get N^2 triads, $(z(u,j),z(v,j),f(u,v))$, where $z(u,j)$ and $z(v,j)$ are the mapping values obtained for the u^{th} and v^{th} images for the j^{th} column through the indexing function, and $f(u,v)$ is the corresponding label
- Potential threshold values are tried and error rates are calculated. The threshold value with the smallest error rate is chosen as the threshold for the present column

Learning thresholds

- Input of N^2 triads $(z(u,j),z(v,j),f(u,v))$. An empty array A and $T_n = 0$, which is a count of the number of negative $f(u,v)$ values
- For each triad :
  If $z(u,j) > z(v,j)$, then $l_1 = 1$
  Else if $z(u,j) < z(v,j)$, then $l_1 = -1$
  Else $l_1 = 0$
  $l_2 = -l_1$
  Append $(z(u,j),l_1,f(u,v))$ and $(z(v,j),l_2,f(u,v))$ to A
  If $f(u,v) == -1$, then T_n++
- Sort A by z values. $s_p = s_n = 0$. $c_b = T_n$ and $T_j = \min(z)$ - epsilon
- For each triad (z,l,f) in A :
  If $f == 1$, then $s_p = s_p - l$
  Else if $f == -1$, then $s_n = s_n - l$
  $c = T_n - s_n + s_p$
  If $c < c_b$, then $c_b = c$ and $T_j = z$

THANK YOU