# Approximating Distances Using LSH

October 27, 2017

## 1 Local Sensitive Hashing

### 1.1 Aproximating distance matrices using LSH

The idea of this notebook is to conduct a series of experiments on using LSH to approximate distance matrices. Theoritically it should be faster, but as with many languages: Python3 probably has distance computations very optimized, on a level that is hard to compete with normal code. Still, here it goes!

```
In [18]: #Imports and a little bit of setup
         from __future__ import print_function
         import matplotlib.pyplot as plt
         from mnist import MNIST

         import math

         #A little bit of matplotlib magic so that the images can be showed on the IPython not
         %matplotlib inline
         plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
         plt.rcParams['image.interpolation'] = 'nearest'
         plt.rcParams['image.cmap'] = 'gray'

         #Small function that helps displays images
         def imshow_noax(img, normalize=True):
             """ Tiny helper to show images as uint8 and remove axis labels """
             if normalize:
                 img_max, img_min = np.max(img), np.min(img)
                 img = 255.0 * (img - img_min) / (img_max - img_min)
             plt.imshow(img.astype('uint8'))
             plt.gca().axis('off')
```

### 1.2 Custom Implementation of LSH

Eventhough there are great libraries out there that implement LSH (FALCONN or lshash 0.04dev), since what we are doing is very specific, it's a better idea to write the procedure ourselves

```
In [ ]: #Scripts that approximates a distance matrix using Local
        #Sensitive Hashing for Hyperplanes
```

```python
import scipy.spatial.distance as distance
import numpy as np
#For product of numpy arrays
import itertools




#Class that does some of the funcionalities of a hash table
class SemiHashTable:
    def __init__(self):
        #The dictionary
        self.dic = {}

    def add_element(self, e, k):
        if(not(k in self.dic)):
            self.dic[k] = []

        self.dic[k].append(e)

    def get_elements(self, k):
        return self.dic[k]

    def get_dictionary(self):
        return self.dic

    def get_all_tuples(self):
        response = []
        for key, value in self.dic.items():
            if(len(value)>1):
                response =  response + list(itertools.combinations(value, r = 2))

        return response

    def get_all_buckets(self):
        response = []
        for key, value in self.dic.items():
            response.append(value)

        return response

#Class used to approximate the distance matrix
class DistanceDictionary:

    def __init__(self, N, unknown_val = np.inf):
        #Number of elements and dimension
        self.N = N
```

```python
        #delta
        self.unknown_val = unknown_val
        #The dictionary
        self.dic = {}

    #Elements should
    def add_pair(self, pair, value ):
        x,y = np.sort(pair)
        if(not(x in self.dic)):
            self.dic[x] = {}

        if(not(y in self.dic[x])):
            self.dic[x][y] = value

    def get_distance(self, pair):
        x,y = np.sort(pair)
        if(x==y):
            return 0
        if(not(x in self.dic)):
            return self.unknown_val
        if(not(y in self.dic[x])):
            return self.unknown_val

        return self.dic[x][y]

    def get_computed_percentage(self):
        #the zero diagonal
        total = self.N
        for key, value in self.dic.items():
            total += 2*len(value)

        return total/(self.N**2)

    def get_distance_matrix(self):

        result = np.zeros((self.N,self.N))

        #Sets the calculated distances
        for i in range(self.N):
            for j in range(i + 1,self.N):
                d = self.get_distance((i,j))
                result[i,j] = d
                result[j,i] = d



        return result
```

```
#Main function
def approximate_distance_matrix(X, K, L, metric_fun = None, hash_function = None, appro
    '''
    Parameters
    -----------------------------
    X : numpy.array
        An NxD numpy array. A set of N observations of dimension D.
    K : integer > 0
        The number of planes that will split the data set.
    L : Integer > 0
        The number of times the splitting procedure will be carried out.
    metric : str or function (optional)
        The string or function to calculate distance between records.
        This parameter will be passed into the 'scipy.spatial.distance.pdist'
        function under the parameter: metric. NOTE: although the user can
        select any distance, this implementation only makes sense for the
        cosine distance, since the hashfuncion to be used is separation
        by hyper planes. The function will should recieve a a numpy array
        of dimension (2,D)
    hash_function : function (optioanl)
        A function that revieves the entire set X of dimension (N,D) and returns
        the resulting hash for every element, a numpy array of dimension (N,1)
    -----------------------------
    Return
    distance_matrix : np.matrix
        A numpy NxN matrix that corresponds to the approximation of
        the actual distance matrix computed under the given metric .
    '''
    #Extracts the dimensions of the input
    N,D = X.shape

    if(metric_fun is None):
        metric_fun = lambda x : distance.pdist(x, metric = cosine)[0]

    if(hash_function is None):
        #Calculates the binary multiplication vector, necessary for the hash calculati
        binary_power = 2**np.arange(K)
        #First calculates the center
        center = np.sum(X, axis = 0)/N
        def hash_function(x):
            hashes = np.dot(X-center,(np.random.rand(D,K) - 0.5))
            hashes = hashes > 0
            hashes = np.dot(hashes,binary_power)


    #Distance matrix will be a dictionary, where dist(i,j) = dic.get_distance((i,j))
```

```
if(approx == 'l'):
    distance_matrix = DistanceDictionary(N,0)
else:
    delta = distance.pdist(
        np.array([np.amax(X, axis = 0),np.amin(X, axis = 0)]),
        metric = metric)[0]
    distance_matrix = DistanceDictionary(N,delta)

for i in range(L):

    print('Started: ' + str(i+1) + ' of ' + str(L))

    #Starts the hash table
    hash_table = SemiHashTable()
    for j in range(N):
        hash_table.add_element(j, hashes[j])

    if(approx == 'l'):
        buckets = hash_table.get_all_buckets()
        for j in range(len(buckets)-1):
            for k in range(j+1,len(buckets)):
                pairs = list(itertools.product(buckets[j],buckets[k]))
                dist = metric_fun(X[pairs[0],:])
                for coord in pairs:
                    temp_dist = distance_matrix.get_distance(coord)
                    value = (temp_dist*i + dist)/(i+1)
                    distance_matrix.add_pair(coord, value)


    else:
        similar = hash_table.get_all_tuples()
        for coord in similar:
            dist = metric_fun(X[coord,:])
            distance_matrix.add_pair(coord, dist)


print('Finished')
return distance_matrix
```

### 1.3 Compare $d_l$ and $d_u$ for MNIST in the $R^{784}$ Representation

The following scheme compares both distance approximations for the $R^{784}$ Representation of the MNIST test set. We use the cosine distance and the Hyperplane hash function

In [102]:
```
#Loads the MNIST set
from mnist import MNIST
#Imports the data and converts it to numpy arrays
mndata = MNIST('../python-mnist/data')
X, labels = mndata.load_testing()

#converts to numpy array
X = np.array(X)
labels = np.array(labels)

subsample_idices = np.random.choice(X.shape[0], 1000)
X = X[subsample_idices,:]
labels = labels[subsample_idices]

#Converts to binary the selcted sample
X = np.around(1-(X/255))

N, D = X.shape
```

In [ ]:
```
d = distance.squareform(distance.pdist(X, metric = 'cosine'))

min_L = 1
max_L = 30
step_L = 4

L_coord = list(range(min_L, max_L+1, step_L))

min_K = 1
max_K = 30
step_K = 4

K_coord = list(range(min_K, max_K+1, step_K))

z_l = np.zeros((len(L_coord),len(K_coord)))
z_l[:] = np.nan

z_u = np.zeros((len(L_coord),len(K_coord)))
z_u[:] = np.nan



#Constructs the d_lower matrix
for i in range(len(L_coord)):
    for j in range(len(K_coord)):
        L = L_coord[i]
        K = K_coord[j]
        dl = approximate_distance_matrix(X,K,L, approx = 'l')
        du = approximate_distance_matrix(X,K,L, approx = 'u')
```

```
            z_l[i,j] = np.linalg.norm(dl.get_distance_matrix() - d, np.inf)
            z_u[i,j] = np.linalg.norm(du.get_distance_matrix() - d, np.inf)
            print('Finished: L = ' +  str(L) + ' K = ' + str(K))
```

### 1.4 Constructing $H_{\mathbb{RP}}$

We now set to construct the hash function $H_{\mathbb{RP}}$ and use it to approximate the angular distance

#### 1.4.1 Spherical Coordinates

We first need to convert the given data into n dimensional spherical coordinates

```
In [93]: #Given a vector, it constructs its corresponding spherical coordinates.
         def compute_shperical_coordinates(vec):
             '''
             Parameters
             -----------------------------
             vec : numpy.array
                 An d dimensional numpy array.
             -----------------------------
             Return
             vec_spherical : numpy.array
                 An d dimensional numpy array with the sperical coordinates
             '''
             #Calculations based on:
             # https://en.wikipedia.org/wiki/N-sphere
             #gets dimension
             D = vec.shape[0]
             if(D < 3):
                 raise ValueError('Only supports conversion of dimension >= 3')
             # Gets r
             r = np.linalg.norm(vec)
             if(r == 0):
                 return(vec)

             #Constructs the upper triangular matrix
             #of the repetead vector
             triang = np.triu((np.tile(vec,(D, 1))))

             #Calculates dneominators
             den = np.dot(triang,vec)
             #square root
             den = np.sqrt(den)
             # Since it is possible that the denominator be zero
             # and in that case the corresponding angle is zero
             # we change arrays so arccos gives zero
             vec_edited = np.copy(vec)
```

```
        vec_edited[den == 0] = 1
        den[den == 0] = 1

        # calculates cosine values
        cosine = np.divide(vec_edited,den)
        #gets angles
        angles = np.arccos(cosine)

        #adjusts the last angle
        if(vec[-2] < 0):
            angles[-2] = 2*np.pi - angles[-2]

        #adds radius
        angles[-1] = r
        #shifts the array for the radius to be the first coor
        angles = np.roll(angles,1)
        return(angles)
```

Computes the shperical coordinate for the MNIST test set (in its $\mathbb{RP}^n$ representation)

```
In [105]: import pandas as pd
          #Loads the proyective coordinates into a numpy array
          proy_coordinates = pd.read_csv(
              '../Multi-Scale Projective Coordinates/multi_scale_coordinates_500_all.csv').as_r

          print('Data loaded')
          X = proy_coordinates
          N, D = X.shape

          x_sphe = np.zeros(X.shape)
          for i in range(N):
              x_sphe[i,:] = compute_shperical_coordinates(X[i,:])

          print('Coordinates computed')
```

```
Data loaded
```

### 1.4.2 $H_{\mathbb{RP}}$

We set out to program the hash function

### 1.4.3 The Angular Distance $d_{\mathbb{RP}}$

Function for the corresponding distance in $\mathbb{RP}^n$

```
In [269]: def rp_distance(sample):
              d = distance.squareform(distance.pdist(sample, metric = 'cosine'))
              return(np.arccos(np.abs(-d+1)))
```

```
In [376]: #Function that calculates the hash values
          def get_hash_values(sample, K = 1, k = None):
              '''
              Parameters
              -----------------------------
              sample : numpy.array
                  An NxD dimensional numpy array where values are assumed
                  in RP^{D-1} in spherical coordinates
              K : positive integer
                  The number of repetitions for the given hash
              k : positive integer
                  The number of coordinates that will be taken into account
              -----------------------------
              Return
              hash_values : numpy.array
                  An Nx(K*k)) dimensional numpy binary array, representing the hash values
              '''
              sphere = np.copy(sample)
              #Removes the first coordinate (radius)
              sphere = sphere[:,1:]

              #final shape
              N, D = sphere.shape

              if(k is None or k > D):
                  k = D

              #Since coordinates are in RP^n, we can restrict the last entry of the
              #SPherical coordinates to be in [0,pi]
              last_col = sphere[:,-1]
              last_col[last_col > np.pi] = last_col[last_col > np.pi] - np.pi

              hash_values = None
              for i in range(K):

                  shpere_proj = sphere
                  if(k < D):
                      sample_dimensions = np.random.choice(D, k)
                      shpere_proj = sphere[:,sample_dimensions]

                  #constructs the corresponding alphas
                  alphas = 0.5*np.pi*np.random.rand(k)
                  #first line
                  first_line = alphas
                  #second line
                  second_line = alphas + np.pi*0.5
                  #Calculates the sectors
                  after_first = shpere_proj >= first_line
```

```
        behind_second = shpere_proj < second_line
        if(hash_values is None):
            hash_values = after_first*behind_second
        else:
            hash_values = np.concatenate((hash_values,after_first*behind_second), ax


    return(hash_values)


#Define the bucket function
def get_buckets(hash_values):
    '''
    Parameters
    -----------------------------
    hash_values : numpy.array
        A logical NxD dimensional numpy array where each row
        corresponds to the hash value for the correpending
        element in that row
    -----------------------------
    Return
    buckets : numpy.array
        An KxN dimensional numpy logical array,
        each row corresponds to a bucket where
        the positive values are the elements inside it
    '''

    #The algorithm fails if there are rows with all false
    #to solve this, we assign a zero column where only
    # all zero rows will have a 1
    hash_values_edites = np.zeros((hash_values.shape[0],hash_values.shape[1]+1))
    hash_values_edites[:,:-1] = np.copy(hash_values)
    hash_values_edites[np.sum(hash_values_edites,axis = 1) == 0,-1] = 1

    #final dimensions
    N, D = hash_values_edites.shape
    #finds the unique rows
    uni = np.unique(hash_values_edites, axis = 0)

    num_buckets = uni.shape[0]
    #gets the affinty for each hash_value and bucket value
    affinity = np.dot(hash_values_edites,np.transpose(uni))
    #finds the corresponding bucket
    bucket_ids = np.argmax(affinity,axis = 1)

    buckets = np.zeros((num_buckets,N))
    for i in range(num_buckets):
        buckets[i,:] = (bucket_ids == i)
```

```
        return(buckets)
```

### 1.4.4 Now Calculate the Approximations

```
In [380]: def get_higher_approx(sample, buckets):
              N, D = sample.shape
              K = buckets.shape[0]
              result = np.zeros((N,N)) + np.pi*0.5
              indices = np.arange(N)
              for i in range(K):
                  ind_1 = indices[buckets[i,:].astype(bool)]
                  sub_sample = sample[ind_1,:]
                  matrix = rp_distance(sub_sample)
                  result[ind_1,ind_1.reshape(len(ind_1),1)]  = matrix
              return(result)

          def get_lower_approx(sample, buckets):

              N, D = sample.shape
              K = buckets.shape[0]
              #representative is the first element of the bucket
              rep =  np.argmax(buckets,axis = 1)
              dist_between_buckets = rp_distance(sample[rep,:])
              indices = np.arange(N)
              result = np.zeros((N,N))
              for i in range(K):
                  for j in range(i +1,K):
                      ind_1 = indices[buckets[i,:].astype(bool)]
                      ind_2 = indices[buckets[j,:].astype(bool)]
                      result[ind_1,ind_2.reshape(len(ind_2),1)] = dist_between_buckets[i,j]
                      result[ind_2,ind_1.reshape(len(ind_1),1)] = dist_between_buckets[j,i]

              return(result)


          def iterative_approx(sample, K = 1, L = 1, k = None, approx_type = 'l'):

              N,D = sample.shape
              if(approx_type == 'l'):
                  print('Lower Matrix Scheme')
                  result = np.zeros((N,N))
                  for i in range(L):
                      hash_values = get_hash_values(sample,K = K, k = k)
                      buckets = get_buckets(hash_values)
                      matrix = get_lower_approx(sample,buckets)
```

```
                result = np.maximum(result,matrix)
                print('Finished: ' + str(i) +' of ' + str(L))
            return(result)
        else:
            print('Higher Matrix Scheme')
            result = np.zeros((N,N)) + np.inf
            for i in range(L):
                hash_values = get_hash_values(sample,K = K, k = k)
                buckets = get_buckets(hash_values)
                matrix = get_higher_approx(sample,buckets)
                result = np.minimum(result,matrix)
                print('Finished: ' + str(i) +' of ' + str(L))
            return(result)
```

### 1.5 LSH Over Projective Coordinates

Here we apply the lsh scheme to the computed coordinates over $\mathbb{RP}^{500}$ of the MNIST data set.

```
In [ ]: import pandas as pd
        import math
        #Loads the proyective coordinates into a numpy array
        proy_coordinates = pd.read_csv(
            '../Multi-Scale Projective Coordinates/multi_scale_coordinates_500_all.csv').as_ma

        N,D = proy_coordinates.shape

        print('Data_loaded')

        complete = rp_distance(proy_coordinates)

        print('Complete Distance Computed')

        K = math.ceil(np.log2(D))
        L = 2
        k = K

        approx_distance_matrix = iterative_approx(proy_coordinates, K = K, L = L, k = k, appro

        print(np.max(np.abs(approx_distance_matrix - complete)))

        print('Approx Distance Matrix Computed')

        np.savetxt("lower_distance_matrix_all_angular.csv", approx_distance_matrix, delimiter=
```

```python
print('Saved File')
```