In this session:

- We'll use a python implementation of locality sensitive hashing for images of 8x8 pixels
- We'll compare lsh with exhaustive search
- We'll see the effect that tuning parameters k and m has over speed and accuracy of the lsh method

1 Introduction

In the following sections we describe a simple implementation of the locality sensitive hashing technique seen in class. In particular, in the file `lsh.py` you can find the class "lsh" implementing this technique for a dataset of 1797 images representing hand-written digits.

Your task is to write simple scripts that make use of this class to test the efficacy and accuracy of lsh for finding nearest neighbors of distorted images in this dataset.

2 The digits dataset

For this lab session we use the *digits* dataset from the file `'images.npy'`. You can load this dataset using `numpy`'s load function:

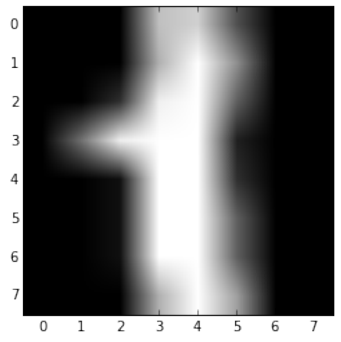
```
data = numpy.load('images.npy')
```

This loads the images into the `data` variable, which is a `numpy` array of 1797 images. Each image is a 8×8 bi-dimensional `numpy` array encoding 64 pixels with integers between 0 and 16. As an example, if one accesses the second image in this list, one gets:

```
>>> data[1]
array([[  0.,   0.,   0.,  12.,  13.,   5.,   0.,   0.],
       [  0.,   0.,   0.,  11.,  16.,   9.,   0.,   0.],
       [  0.,   0.,   3.,  15.,  16.,   6.,   0.,   0.],
       [  0.,   7.,  15.,  16.,  16.,   2.,   0.,   0.],
       [  0.,   0.,   1.,  16.,  16.,   3.,   0.,   0.],
       [  0.,   0.,   1.,  16.,  16.,   6.,   0.,   0.],
       [  0.,   0.,   1.,  16.,  16.,   6.,   0.,   0.],
       [  0.,   0.,   0.,  11.,  16.,  10.,   0.,   0.]])
```

which corresponds to the following image representing the digit 1

```
>>> import matplotlib.pyplot as plt
>>> plt.imshow(data[1], cmap=pylab.cm.gray)
>>> plt.show()
```



3 LSH for the digits dataset

To represent images as strings of bits, we view each image as a flat array of 64 integers (the "pixels"), each with a value between 0 and 16. We represent each pixel as a bitstring of length 16 using the unary encoding. For example, the second image shown above is encoded in binary as the bitstring (spaces are only placed for clarity, and only the first row of pixels is shown):

```
0000000000000000 0000000000000000 0000000000000000 1111111111110000
1111111111111000 1111100000000000 0000000000000000 0000000000000000
...
```

This bitstring is never constructed, but it is useful to keep in mind that we represent images this way. Therefore, each image is represented with a bitstring of length 64x16 = 1024.

Following the technique seen in class, we use bit projections of these strings to construct our hash codes. Namely, we select bits uniformly at random (from 0 to 1023) as our hashing functions. To "amplify the gap" we select k bits randomly from 0 to 1023, and repeat this m times.

Therefore, when our class is initialized, the first thing we do is to create an $m \times k$ matrix with the bits selected uniformly at random. This can be found in the `__init__` function:

```
    def __init__(self, k, m):
        """ k is nr. of bits to hash and m is reapeats """
        # data is numpy ndarray with images
        self.data = numpy.load('images.npy')
        self.k = k
        self.m = m

        # determine length of bit representation of images
        # use conversion from natural numbers to unary code for each pixel,
        # so length of each image is imlen = pixels * maxval
        self.pixels = 64
        self.maxval = 16
        self.imlen = self.pixels * self.maxval

        # need to select k random hash functions for each repeat
        # will place these into an m x k numpy array
        numpy.random.seed(12345)
        self.hashbits = numpy.random.randint(self.imlen, size=(m, k))

        # the following stores the hashed images
        # in a python list of m dictionaries (one for each repeat)
        self.hashes = [dict() for _ in range(self.m)]

        # now, fill it out
        self.hash_all_images()

        return
```

The hashed images are stored into a list of m dictionaries. Notice that only the first 1500 images are hashed and stored, since we are going to use the last 279 remaining ones for testing the technique.

```
    def hash_all_images(self):
        """ go through all images and store them in hash table(s) """
        # Achtung!
        # Only hashing the first 1500 images, the rest are used for testing
        for idx, im in enumerate(self.data[:1500]):
            for i in range(self.m):
                str = self.hashcode(im, i)

                # store it into the dictionary..
                # (well, the index not the whole array!)
                if str not in self.hashes[i]:
                    self.hashes[i][str] = []
                self.hashes[i][str].append(idx)
        return
```

The function **hashcode** computes the hash code for the i'th repeat of a given image (viewed as a bitstring). In order to do that, one needs to compute the projection given by the i'th row of the hashbit matrix of the bit string representing the given image.

```
    def hashcode(self, im, i):
        """ get the i'th hash code of image im (0 <= i < m)
            notice 'im' is the image itself, *not* the index.
        """
        pixels = im.flatten()
        row = self.hashbits[i]
        str = ""
        for x in row:
            # get bit corresponding to x from image..
            pix = int(x) // int(self.maxval)
            num = x % self.maxval
            if num <= pixels[pix]:
                str += '1'
            else:
                str += '0'
        return str
```

Finally, we provide the **candidates** function which, given an image, returns a set of possible nearest neighbors (namely, those image indices that fall into the same "bucket" as the input image for some of the m repeats)

```
    def candidates(self, im):
        """ given image im, return set of indices of matching candidates """
        res = set()
        for i in range(self.m):
            code = self.hashcode(im, i)
            if code in self.hashes[i]:
                res.update(self.hashes[i][code])
        return res
```

4 Your tasks

4.1 Task 1: Understanding the code and basic working of lsh

In the "main" section of the `lsh.py` file, we have included the following code:

```
@timeit
def main(argv=None):
    parser = argparse.ArgumentParser()
    parser.add_argument('-k', default=20, type=int)
    parser.add_argument('-m', default=5, type=int)
    args = parser.parse_args()

    print("Running lsh.py with parameters k =", args.k, "and m =", args.m)

    me = lsh(args.k, args.m)

    # show candidate neighbors for first 10 test images
    for r in range(1500, 1510):
        im = me.data[r]
        cands = me.candidates(im)
        print("there are %4d candidates for image %4d" % (len(cands), r))

    return
```

Make sure you understand the code, and invoke this script with different input parameters (vary k and m).

- The running time of the hashing process is $O(km)$ in theory, by looking at the code. Does this agree with experiments?
- What happens to the size of the candidate set when increasing k? And when you increase m? Can you explain why?
- Can one give a function $f(k, m)$ expressing the size of the candidate set? (or not?)

There are a couple of Pythoneries, one regarding the `timeit` function and the its use as decorator `@timeit`, and the other at the end about `main` being invoked. They are not essential for this lab or have anything to do with lsh, but read and learn about them here if interested:

- `https://medium.com/pythonhive/python-decorator-to-measure-the-execution-time-of-methods-fa0`
- `https://es.stackoverflow.com/questions/32165/qu%C3%A9-es-if-name-main`

4.2 Task 2: Does lsh work?

Now it is time for you to start writing your own code. In this task, you are asked to compare lsh to brute-force search. Namely, you have to:

- Write a function that computes the *distance* between two input images. Use the l_1-distance, i.e., the sum of absolute values of differences between pixels in the input images (which, by the way, coincides with the Hamming distance for the representation of images we have chosen).
- Write a function that implements brute-force search, namely, given an input image, compute its distance to all images in the dataset (well, the first 1500) and outputs the one that is closest to it together with the l_1-distance value.
- Write a search function that, given an input image, uses the hashes to find the nearest neighbor. Namely, given an input image, invoke the `candidates` function provided, and return the closest image

in the candidate set (together with the l_1 distance). Notice that depending on the parameters k, m that you chose, you may have no candidates; in this case, your function should return `None` (python's null value).

Once you have implemented all these, try to measure how long it takes to find the nearest neighbor with both methods (brute force and lsh), and also the distance between the input image and the ones returned by brute-force and by lsh. You can play with the k and m parameters to find satisfactory answers.

5 Deliverables

Rules: Same rules as in previous labs apply.

To deliver: You must deliver 1) the program file or files you wrote and 2) a brief report (2 pages at the most) explaining the main difficulties/choices you had in implementing the scheme and any comments you have on the process or the result.

Procedure: Submit your work through the Racó as a single zipped file.

Deadline: Work must be delivered within **2 weeks** from the lab. Late deliveries risk being penalized or not accepted at all. If you anticipate problems with the deadline, tell me as soon as possible.