828J Project 2

Theme: Experimentation with locality-sensitive hashing for approximate nearest neighbors classification.

We implement two classifiers for approximate nearest neighbor classification. The first classifier is a standard R-near neighbor classifier. The second one is a locality-sensitive hashing (LSH) classifier, where the LSH function family $\mathcal{H}$ is drawn from the family of random projections onto $\mathbf{R}$. We tune both classifiers to find optimal parameters, such as the distance radius R for the original classifier and the bucket width w for the new LSH classifier. Experiments on the "Gisette" optical character recognition dataset show that our locality-sensitive classifier:

- Achieves comparable accuracy to the non-locality sensitive classifier after tuning for various parameters,
- Is considerably faster at test time, because of the nature of the underlying data structure, and
- Is considerably slower at training time, because a standard nearest neighbor classifier need only copy the training data, whereas a locality-sensitive classifier needs to build a complex data structure.

Our goal with this programming project is not to go through an exhaustive comparison of simple approximate nearest neighbor (ANN) classifiers and LSH ANN classifiers, but to highlight the three key points described above.

Documentation format

This PDF has been generated using the iPython notebook, a software which helps integrate Python code snippets with their respective outputs. The gray cells are code snippets, and the typewriter font-like text underneath them is the output of these code snippets, if any. The white cells, like the current one, are MarkDown (text) snippets. The iPython notebook is similar to MATLAB's cell-based HTML/PDF export feature. Our goal has been to make it possible for the reader not to read the code at all: the implementation can be understood in its entirety by reading through this PDF.

List of submitted files

- input_data: The "Gisette" dataset (downloaded via HTTP from http://archive.ics.uci.edu/ml/machine-learning-databases/gisette/GISETTE/)
- proc_data: Some intermediate binary files that are the result of input data pre-processing, and are used by our code.
- output_data: Binary and/or textual output data produced by our scripts.
- *.py, *.pyc Python source (*.py) and class (*.pyc) files.
- README.ipynb: The original iPython notebook which generated the current PDF.
- README.pdf: The current PDF.
- LICENSE: The AFL 3.0 academic license text

The ANN and LSH_ANN classifiers

Code and LSH data structure

Our approximate nearest neighbor classifier (ANN) has been implemented as a Python class in ANN.py. It involves methods for training, tuning and testing on subsets of data. The parameters of the classifier are the radius R for finding R-near neighbors and the number of maximum R-near neighbors N to consider during voting for a test point label.

Our locality-sensitive approximate nearest neighbor classifier (LSH_ANN) has been implemented in the python class LSH_ANN.py. It involves its own methods for training, tuning and testing. Tuneable parameters include the quantization bin width w and the maximum number N of approximate nearest neighbors to consider at test time. Briefly, the underlying data structure is a list of L hash tables, each of which has k-dimensional tuples as keys and training point indices (integers) as mapped values. The k-dimensional keys are the result of projecting every training point randomly onto R k times. This is accomplished by considering,

for every hash table, k different D-dimensional vectors whose components are drawn i.i.d from a multi-variate Gaussian distribution.

Advantages and disadvantages of either classifier over the other

The main benefit of ANN over LSH_ANN is that it practically has no training time overhead, because a standard nearest neighbor classifier's training phase consists of simply storing the training data. LSH_ANN, on the other hand, has to build the entire locality-sensitive hashing data structure at training time. On the other hand, ANN's testing phase is computationally expensive: It loops through the entire training data and measures Euclidean distance metrics, which is also a problem considering the potentially enormous dimensionality of the space (for Gisette, there are 5.000 dimensions, including "probe" features).

One thing we find to be underestimated in the LSH literature that we have gone through is that an LSH classifier is hard to tune. This is because, in an air-tight application, such a classifier should have the following 4 parameters tuned:

- The quantization bin width, w.
- The number of hash tables, L.
- The number of random projections per hash table, k.
- The number of R-near neighbors to consider, N

The reader could imagine the tuning for all these parameters occurring via a - possibly multi-threaded - quadruple "for loop" of some width, and if a tuning procedure via 10-fold cross validation is envisioned, this entire process will have to be repeated 10 times. In addition, in order to tune for w, L and k, the entire classifier needs to be re-trained. This is bad news for a locality-sensitive classifier, because, compared to a non-locality sensitive classifier, its training phase is much more computationally intensive.

Given computational and time constraints, we were not able to do a thorough search of the parameter space. For example, for the number of hash tables L and number of random projections k we used the values suggested by Dr. Jacobs (L = 80, k = 15) in class.

Experimental setup

(a) Dataset

The "Gisette" optical character recognition dataset (http://archive.ics.uci.edu/ml/datasets/Gisette) aims to help researchers with developing algorithms that can discern between the - often confusible - handwritten digits 4 and 9. The dataset offers labels for training and development data. The testing data labels were withheld, because the dataset was part of the NIPS 2003 feature selection challenge, where challenge participants were supposed to submit their test data results to the organizers via a backend server. Gisette offers a total of 6000 training examples (3000 of each class), 1000 development examples (500 of each class) and 6500 testing examples (3250 examples of each class, but with the labels withheld). The feature vectors are 5000-dimensional and included a number of "probe" (i.e uninformative) features, such that the best algorithms were the ones who were able to filter out the probes. The dataset itself, as well as a ".params" text file with statistics and MD5 hashes, can be found under the "input_data" subdirectory.

(b) Preparation

We use the same training data for both classifiers. We split the development data into two subsets: validation data and testing data. We do this via random permutation of the examples, but for the purposes of our experiments we used a static seed of 47 for the random number generator, such that our results can be reproduced. The reader can simply erase the line "np.random.seed(47)" at the start of the main methods in the code provided in order to re-produce the experiments in a fully (pseudo-)random setting.

The following code snippet will load the data in memory:

In [1]:
```
import pickle as pkl
import time as t
import numpy as np
from ANN import ANN
from LSH_ANN import LSH_ANN

if __name__ == "__main__":

        np.random.seed(47) # To make results reproducible

        # Step 1: Read the stored data
```

```
trainData = pkl.load(open('proc_data/trainDat.pkl'))
trainLabels = pkl.load(open('proc_data/trainLabs.pkl'))
validData = pkl.load(open('proc_data/validDat.pkl'))
validLabels = pkl.load(open('proc_data/validLabs.pkl'))
print "Data loaded!"

# Step 2: Split the validation data into actual validation and
# testing data

indices = [index for (index, _) in list(enumerate(validLabels))]
indices = np.random.permutation(indices)
validIndices = indices[0:500]
testIndices = indices[500:1000]
validData_new = validData[validIndices]
validLabels_new = validLabels[validIndices]
testData = validData[testIndices]
testLabels = validLabels[testIndices]
print "Split performed!"
print "#Validation examples: " + str(len(validData_new))
print "#Testing examples: " + str(len(testData))
```

```
Data loaded!
Split performed!
#Validation examples: 500
#Testing examples: 500
```

Tuning the classifiers

As previously mentioned, there are a number of parameters that need to be tuned for both classifiers. For the ANN classifier, the range of the radii R that we tuned over was determined empirically to be the set [20000, 20100, ..., 21000], after building the Gram matrix of the training data points and doing some basic statistical analysis. The following code cell tunes for both the radius R and the number of nearest neighbors N.

In [13]:
```
from classification_utils import train, tune, test

for R in range(20000, 21000, 100):  # Test for 10 different radii
    ANNClassifier = train(trainData, trainLabels,R, None)
    print "Trained ANN classifier for R = " + str(R)
    optimalN = tune(ANNClassifier, validData_new, validLabels_new, range(1, 11))
    print "The optimal value of N found for R = " + str(R) + " was: " + str(optimalN)
```

```
Trained ANN classifier for R = 20000
The optimal value of N found for R = 20000 was: 6
Trained ANN classifier for R = 20100
The optimal value of N found for R = 20100 was: 4
Trained ANN classifier for R = 20200
The optimal value of N found for R = 20200 was: 5
Trained ANN classifier for R = 20300
The optimal value of N found for R = 20300 was: 9
Trained ANN classifier for R = 20400
The optimal value of N found for R = 20400 was: 2
Trained ANN classifier for R = 20500
The optimal value of N found for R = 20500 was: 8
Trained ANN classifier for R = 20600
The optimal value of N found for R = 20600 was: 9
Trained ANN classifier for R = 20700
The optimal value of N found for R = 20700 was: 6
```

```
Trained ANN classifier for R = 20800
The optimal value of N found for R = 20800 was: 9
Trained ANN classifier for R = 20900
The optimal value of N found for R = 20900 was: 6
```

We tuned the LSH_ANN classifier under two different parameters. First, we determined a value of the quantization bin width w which minimized validation error. The error was measured by voting over all R-near neighbors of a test point. We used the following code snippet to do this:

In [12]:
```
### Tune the classifier for w
for w in range(4, 10):
    classifier = LSH_ANN(80, 15, w)
    classifier.train(trainData, trainLabels)
    accuracy = classifier.test(validData_new, validLabels_new)
    print "Attained an accuracy of: " + str(accuracy) + " for a bucket width of " + s
```

```
Randomly voted 0 times.
Attained an accuracy of: 0.502 for a bucket width of 4
Randomly voted 0 times.
Attained an accuracy of: 0.476 for a bucket width of 5
Randomly voted 0 times.
Attained an accuracy of: 0.488 for a bucket width of 6
Randomly voted 0 times.
Attained an accuracy of: 0.484 for a bucket width of 7
Randomly voted 0 times.
Attained an accuracy of: 0.496 for a bucket width of 8
Randomly voted 0 times.
Attained an accuracy of: 0.532 for a bucket width of 9
```

We note that the best accuracy was detected for a bucket width of 9. Therefore, when tuning for the optimal number of neighbors, we used this bucket width. To perform this tuning, we switched the above for loop with the following for loop to discover the effect of limiting the number of neighbors to a maximum of 10:

In [14]:
```
### Tune the classifier for L' (number of R-near neighbors) while keeping a bin widtl
for n in range(1, 11, 1):
    classifier = LSH_ANN(80, 15, 9)
    classifier.train(trainData, trainLabels)
    accuracy = classifier.test(validData_new, validLabels_new)
    print "Attained an accuracy of: " + str(accuracy) + " for a number of neighbors =
```

```
Randomly voted 0 times.
Attained an accuracy of: 0.506 for a number of neighbors = 1 and a bucket width of
9.
Randomly voted 0 times.
Attained an accuracy of: 0.506 for a number of neighbors = 2 and a bucket width of
9.
Randomly voted 0 times.
Attained an accuracy of: 0.516 for a number of neighbors = 3 and a bucket width of
9.
Randomly voted 0 times.
Attained an accuracy of: 0.53 for a number of neighbors = 4 and a bucket width of 9.
Randomly voted 0 times.
Attained an accuracy of: 0.498 for a number of neighbors = 5 and a bucket width of
9.
Randomly voted 0 times.
Attained an accuracy of: 0.536 for a number of neighbors = 6 and a bucket width of
9.
Randomly voted 0 times.
Attained an accuracy of: 0.504 for a number of neighbors = 7 and a bucket width of
```

```
9.
Randomly voted 0 times.
Attained an accuracy of: 0.512 for a number of neighbors = 8 and a bucket width of
9.
Randomly voted 0 times.
Attained an accuracy of: 0.482 for a number of neighbors = 9 and a bucket width of
9.
Randomly voted 0 times.
Attained an accuracy of: 0.484 for a number of neighbors = 10 and a bucket width of
9.
```

So the best accuracy was detected for a number of neighbors equal to 6. It appears that limiting the neighbors actually aided classification accuracy in this case. This is not too surprising, because even in classic ANN classifiers considering too many neighbors can have a detrimental effect, especially when talking about data on the edge of a cluster.

Test-time comparison

The tuning process hopefully convinced the reader that it is possible to achieve comparable accuracy by projecting all the points randomly onto R by using a locality- sensitive classifier. We will use the parameters found to be optimal for each classifier by the tuning process and measure both training time and testing time. We achieve this via the following code snippet:

In [7]:
```
concatData = np.concatenate([trainData, validData_new])
concatLabs = np.concatenate([trainLabels, validLabels_new])


ANN_cl = ANN(20600, 7)
LSH_ANN_cl = LSH_ANN(80, 15, 9)

start = t.time()        # milliseconds since epoch
ANN_cl.train(concatData, concatLabs)
end = t.time()
print "It took us " + str(end - start) + " milliseconds to train the ANN classifier."

start = t.time()
LSH_ANN_cl.train(concatData, concatLabs)
end = t.time()
print "It took us " + str(end - start) + " milliseconds to train the LSH_ANN classif

start = t.time()
ANN_cl_acc = ANN_cl.test(testData, testLabels, None)
end = t.time()
print "On the testing data, the ANN classifier achieved an accuracy of: " + str(ANN_c
print "It took it " + str(end - start) + " milliseconds to test the data."

start = t.time()
LSH_ANN_cl_acc = LSH_ANN_cl.test(testData, testLabels, 6)
end = t.time()

print "On the testing data, the LSH_ANN classifier achieved an accuracy of: " + str(L
print "It took it " + str(end - start) + " milliseconds to test the data."
```

```
It took us 0.0 milliseconds to train the ANN classifier.
It took us 75.631000042 milliseconds to train the LSH_ANN classifier.
For radius = 20600, points found alone: 121
On the testing data, the ANN classifier achieved an accuracy of: 0.496.
It took it 247.351000071 milliseconds to test the data.
Randomly voted 0 times.
```

```
Randomly voted 0 times.

On the testing data, the LSH_ANN classifier achieved an accuracy of: 0.494.
It took it 6.64299988747 milliseconds to test the data.
```

The ANN classifier takes less than 0 milliseconds to train, since it implies a simple reference copy.(*) The LSH_ANN classifier, on the other hand, takes more than 75 times that amount of time, owing to the nature of the data structure. However, at testing time, the LSH_ANN classifier improves upon the execution time of the non-locality sensitive classifier by a factor of about 37.

(*) We ran the "time()" function in Windows, where the accuracy is limited to milliseconds from the Epoch. The Linux implementation is more fine-grained, leading to nanosecond accuracy.

Conclusions

The benefit to be had by using an approximate nearest neighbor classifier which works with locality sensitive hashing lies on the test-time speed. Our final experimentation shows that the LSH_ANN classifier tests the same amount of data several orders of magnitude faster than the ANN classifier, while simultaneously exhibiting only .2% smaller accuracy. It should also be noted that, in terms of the common dimensionality encountered in Computer Vision datasets, Gisette is infant-like; typical Vision datasets have dimensionalities which range to hundreds of thousands of dimensions, if not millions. In the latter cases, the benefit of using an LSH-ANN classifier instead of an ANN classifier is maximized.

Tuning for the parameters L, k and w of an LSH_ANN classifier yields a trade-off between classification accuracy and test-time speed. The probability that two R-near neighbors are hashed to the same bucket (and, conversely, two points that are NOT R-near neighbors being hashed to DIFFERENT buckets) is controlled in its entirety from these parameters. We note that, by doing no tuning at all for either L and k, we were able to achieve an accuracy of only .2% less than the original ANN classifier.

Unfortunately, the practical application of a locality-sensitive hashing classifier seems open to question. This is because of the amount of parameters that need to be tuned, as well as the fact that its training phase is orders of magnitude slower than that of a standard approximate nearest neighbor classifier, which only needs to copy references or pointers to the data. A practical implementation of a locality-sensitive classifier would be highly multi-threaded and vectorized. While our implementation strives for some vectorization by using Python's list comprehensions, it is single-threaded. Moreover, the experiments were run on a laptop which had to run numerous other CPU-intensive applications in parallel, because of our grad studies requirements. Even after a more concrete implementation, the classifier would still be mostly useful in an online non-adaptive setting (i.e a task where new test points come at frequent time intervals, yet the classifier doesn't need to be re-trained). Investigating the possibility of online learning of locality-sensitive hashing classifiers might yield some insights into how this problem could be somehow leveraged.