Improving performance of Local outlier factor with KD-Trees

Local outlier factor (LOF) is an outlier detection algorithm, that detects outliers based on comparing local density of data instance with its neighbors. It does so to decide if data instance belongs to region of similar density. It can detect an outlier in a dataset, for which number of clusters is unknown, and clusters are of different density and size. It’s inspired from KNN (K-Nearest Neighbors) algorithm, and is widely used. There is a R implementation available.

The naive approach to do this is to form all pair euclidian distance matrix, and then run knn query to proceed further. But this approach just sucks, as it is $\Theta(n^2)$ in terms of both space and time complexity. But, this can be improvd with KD-Trees, and already its implementation exists in python, thanks to scipy, so lets use this to find outliers.

Synthetic dataset

```python
In [229]: %pylab inline
import numpy as np
np.random.seed(2) # to reproduce the result

Populating the interactive namespace from numpy and matplotlib

WARNING: pylab import has clobbered these variables: ['dist']
'%.pylab --no-import-all' prevents importing * from pylab and numpy

In [230]: dim = 2 # number of dimensions of dataset = 2
   # cluster of normal random variable moderately dense
    data1 = np.random.multivariate_normal([0, 1500], [[100000, 0], [0, 100000]], 2000)

    # very dense
    data2 = np.random.multivariate_normal([2000, 0], [[10000, 0], [0, 10000]], 2500)

    # sparse
    data3 = np.random.multivariate_normal([2500, 2500], [[100000, 0], [0, 100000]], 500)

    # mix the three dataset and shuffle
    data = np.vstack((np.vstack((data1, data2)), data3))
    np.random.shuffle(data)

    # add some noise : zipf is skewed distribution and can have extreme values(outliers)
    zipf_alpha = 2.25
    noise = np.random.zipf(zipf_alpha, (5000,dim)) * np.sign((np.random.randint(2, size = (5000, dim)) - 0.5))
    data += noise
```

```
Naive approach to LOF

Pairwise Euclidean distance calculation with DistanceMetric implementation in scikit-learn. In this, we just compute all-pair euclidean distance, i.e. $d(i, j) = \| x(i) - x(j) \|_2$.

In [231]: from sklearn.neighbors import DistanceMetric
   #: distance between points
   import time
   tic = time.time()
   dist = DistanceMetric.get_metric('euclidean').pairwise(data)
   print '++ took %g msecs for Distance computation' % ((time.time() - tic)* 1000)
++ took 740 msecs for Distance computation

Performing KNN query. In this step, the nearest k neighbors are identified $N_k(i)$, and radius is the distance of k-th nearest neighbor of a datapoint.

$$r(i) = \max_{k \in N_k(i)} d(i, k)$$

In [232]: tic = time.time()
   #: number of neighbors to consider
   k = 17
   # get the radius for each point in dataset (distance to kth nearest neighbor)
   # radius is the distance of kth nearest point for each point in dataset
   idx_knn = np.argsort(dist, axis=1)[:,1 : k + 1] # by row' get k nearest neighbour
   radius = np.linalg.norm(data - data[idx_knn[:, -1]], axis = 1) # radius
   print '+++ took %g msecs for KNN Querying' % ((time.time() - tic)* 1000)
+++ took 4800 msecs for KNN Querying

Then LRD(Local Reachability distance) is calculated. For this, first reach distance $rd(i, j)$ is computed between point concern $x(i)$ and its neighbors $j : j \in N_k(i)$, which is the maximum of euclidean distance or radius $r(i)$ of point concerned. Then, $LRD$ is the inverse of mean of reach distance of all $k$ neighbors of each point.

$$rd(i, j) = \max\{d(i, j), r(i)\}$$
$$LRD(i) = \frac{|N_k(i)|}{\sum_{j \in N_k(i)} rd(i, j)}$$

In [233]: # calculate the local reachability density
   tic = time.time()
   LRD = []
   for i in range(idx_knn.shape[0]):
       LRD.append(np.mean(np.maximum(dist[i, idx_knn[i]], radius[idx_knn[i]])))
   print '++++ took %g msecs for LRD computation' % ((time.time() - tic)* 1000)
++++ took 429 msecs for LRD computation

finally, the outlier score $LOF$ is calculated.

$$LOF(i) = \frac{\sum_{j \in N_k(i)} LRD(j)}{|N_k(i)|}$$

In [234]: # calculating the outlier score
   tic = time.time()
   rho = 1. / np.array(LRD) # inverse of density
   outlier_score = np.sum(rho[idx_knn], axis = 1)/ np.array(rho, dtype = np.float16)
   outlier_score *= 1./k
   print '+++++ took %g msecs for Outlier scoring' % ((time.time() - tic)* 1000)
+++++ took 577 msecs for Outlier scoring
Now let's see the histogram of Outlier score, to choose the optimal threshold to decide whether a data-point is outlier or not.

In [235]: weights = np.ones_like(outlier_score)/outlier_score.shape[0] # to normalize the histogram to probability plot
hist(outlier_score, bins = 50, weights = weights, histtype = 'stepfilled', color = 'cyan')
title('Distribution of outlier score')

Out[235]: <matplotlib.text.Text at 0x36030588>

It can be observed that, the optimal outlier score threshold to decide whether a data-point is outlier or not is around 2 for most of the cases, so let's use it to see our results.

In [236]: threshold = 2.
   # plot non outliers as green
scatter(data[:, 0], data[:, 1], c = 'green', s = 10, edgecolors='None', alpha=0.5)
   # find the outliers and plot the outliers
idx = np.where(outlier_score > threshold)
s = 10, edgecolors='None', alpha=0.5)
scatter(data[idx, 0], data[idx, 1], c = 'red', s = 10, edgecolors='None', alpha=0.5)

Out[236]: <matplotlib.collections.PathCollection at 0x3640e6a0>
We have seen the results of LOF with naive approach for KNN queries. Now let's see optimisations with KD-Trees.

**Using KD Trees**  KD-Trees insertion and KNN query.

**In [239]:** from sklearn.neighbors import KDTree as Tree  
    tic = time.time()  
    BT = Tree(data, leaf_size=5, p=2)  
    # Query for k nearest, k + 1 because one of the returnee is self  
    dx, idx_knn = BT.query(data[:, :], k = k + 1)  

    print '++ took %g msecs for Tree KNN Querying' % ((time.time() - tic)* 1000)

++ took 122 msecs for Tree KNN Querying

LRD computation.

**In [240]:** tic = time.time()  
    dx, idx_knn = dx[:, 1:], idx_knn[:, 1:]  
    # get the radius for each point in dataset  
    # radius is the distance of kth nearest point for each point in dataset  
    radius = dx[:, -1]  
    # calculate the local reachability density  
    LRD = np.mean(np.maximum(dx, radius[idx_knn]), axis = 1)  

    print '++ took %g msecs for LRD computation' % ((time.time() - tic)* 1000)

++ took 8.99982 msecs for LRD computation

Now, rest is same, so, I'm just replicating the result for completion.
In [241]: # calculating the outlier score
tic = time.time()
rho = 1. / np.array(LRD) # inverse of density
outlier_score = np.sum(rho[idx_knn], axis = 1)/ np.array(rho, dtype = np.float16)
outlier_score *= 1./k
print '+++++ took %g msecs for Outlier scoring' % ((time.time() - tic)* 1000)

# plotting the histogram of outlier score
weights = np.ones_like(outlier_score)/outlier_score.shape[0] # to normalize the histogram to probability
hist(outlier_score, bins = 50, weights = weights, histtype = 'stepfilled', color = 'cyan')

# plotting the result
threshold = 2.
# plot non outliers as green
figure() 
scatter(data[:, 0], data[:, 1], c = 'green', s = 10, edgecolors='None', alpha=0.5)
# find the outliers and plot te outliers
idx = np.where(outlier_score > threshold)
scatter(data[idx, 0], data[idx, 1], c = 'red', s = 10, edgecolors='None', alpha=0.5)

+++++ took 4.00019 msecs for Outlier scoring

Out[241]: <matplotlib.collections.PathCollection at 0x36ad0b38>
The results are same, and should be.

**Putting everything together**  Let's create a class, to combine everything together. It will be important in evaluating performance. From above results, we note that the most time is spent for KNN querying.

In [225]:
```python
import numpy as np
import matplotlib.pyplot as plt
import sys
from sklearn.neighbors import DistanceMetric
from sklearn.datasets import make_blobs
from sklearn.neighbors import KDTree as Tree

def exit():
    sys.exit()

class LOF:
    def __init__(self, k = 3):
        self.k = k

    # a function to create synthetic test data
    def generate_data(self, n = 500, dim = 3):
        n1, n2 = n / 3, n / 5
        n3 = n - n1 - n2

        # cluster of gaussian random data
        data1, _ = make_blobs(n1, dim, centers=3)

        # cluster of uniform random variable
        data2 = np.random.uniform(0, 25, size=(n2, dim))
```
# cluster of dense uniform random variable

data3 = np.random.uniform(100, 200, size = (n3, dim))

# mix the three dataset
self.data = np.vstack((np.vstack((data1, data2)), data3))
np.random.shuffle(self.data)

# add some noise : zipf is skewed distribution
zipf_alpha = 2.5
noise = np.random.zipf(zipf_alpha, (n,dim)) * np.sign((np.random.randint(2, size = (n, dim)) - 0.5))
self.data += noise

# KNN querying with naive approach
def _knn_naive(self):
    # distance between points
    # import time
    tic = time.time()
    dist = DistanceMetric.get_metric('euclidean').pairwise(self.data)
    # print '++ took %g msecs for Distance computation' % ((time.time() - tic)* 1000)
    tic = time.time()
    # get the radius for each point in dataset (distance to kth nearest neighbor)
    # radius is the distance of kth nearest point for each point in dataset
    self.idx_knn = np.argsort(dist, axis=1)[:, 1 : self.k + 1]  # by row get k nearest neighbors
    radius = np.linalg.norm(self.data - self.data[self.idx_knn[:, -1]], axis = 1)  # radius
    # print +++ took %g msecs for KNN Querying' % ((time.time() - tic)* 1000)
    # calculate the local reachability density
    LRD = []
    for i in range(self.idx_knn.shape[0]):
        LRD.append(np.mean(np.maximum(dist[i, self.idx_knn[i]], radius[self.idx_knn[i]])))
    print('+++ took %g msecs for KNN Querying' % ((time.time() - tic) * 1000))
    return np.array(LRD)

# KNN querying with KDTree
def _knn_tree(self):
    # import time
    # tic = time.time()
    BT = Tree(self.data, leaf_size=5, p=2)

    # Query for k nearest, k + 1 because one of the returnees is self
    dx, self.idx_knn = BT.query(self.data[:, :], k = self.k + 1)
    # print '++ took %g msecs for Tree KNN Querying' % ((time.time() - tic)* 1000)
    dx, self.idx_knn = dx[:, 1:], self.idx_knn[:, 1:]
    # get the radius for each point in dataset
    # radius is the distance of kth nearest point for each point in dataset
    radius = dx[:, -1]
    # calculate the local reachability density
    LRD = np.mean(np.maximum(dx, radius[self.idx_knn]), axis = 1)
    return LRD
```python
def train(self, data = None, method = 'Naive') :
    # check if dataset is provided for training
    try:
        assert data != None and data.shape[0]
        self.data = data
        n = self.data.shape[0]  # number of data points
    except AssertionError:
        try:
            n = self.data.shape[0]  # number of data points
        except AttributeError:
            print 'No data to fit the model, please provide data or call generate_data method'
        exit()
    except AssertionError:
        print 'Method must be Naive|n or tree|t'
        exit()

    # find the rho, which is inverse of LRD
    if method.lower() in ['naive', 'n']:
        rho = 1./ self._knn_naive()
    elif method.lower() in ['tree', 't']:
        rho = 1./ self._knn_tree()

    self.score = np.sum(rho[self.idx_knn], axis = 1)/ np.array(rho, dtype = np.float16)
    self.score *= 1./self.k

def plot(self, threshold = None):
    # set the threshold
    if not threshold:
        from scipy.stats.mstats import mquantiles
        threshold = max(mquantiles(self.score, prob = 0.95), 2.)
        self.threshold = threshold
    # reduce data to 2D if required
    if self.data.shape[1] > 2:
        from sklearn.decomposition import PCA
        pca = PCA(n_components = 2)
        self.data = pca.fit_transform(self.data)

    # plot non outliers as green
    plt.figure()
    plt.scatter(self.data[:, 0], self.data[:, 1], c = 'green', s = 10, edgecolors='None', alpha=0.5)

    # find the outliers and plot te outliers
    idx = np.where(self.score > self.threshold)
    plt.scatter(self.data[idx, 0], self.data[idx, 1], c = 'red', s = 10, edgecolors='None')
    plt.legend(['Normal', 'Outliers'])

    # plot the distribution of outlier score
    plt.figure()
```

weights = np.ones_like(self.score)/self.score.shape[0]
plt.hist(self.score, bins = 25, weights = weights, histtype = 'stepfilled', color = 'cyan'
plt.title('Distribution of outlier score')

Performance Evaluation  Lets create a function to evaluate te performance.

In [226]: def perf_test(n_list = None, methods = ['Tree', 'Naive'], plot = False):

    import time
    if not n_list: n_list = [2 ** i for i in range(7, 14)]
    result = []
    result.append(n_list)
    for m in methods:
        temp = []
        for n in n_list:
            tic = time.time()
            lof = LOF(k = 5)
            lof.generate_data(n = n, dim = 2)
            lof.train(method = m)
            temp.append(1000000 * (time.time()-tic))
            print 'Took %g msecs with %s method for %d datapoints' % 
                ((time.time() - tic) * 1000, m, n)
        result.append(temp)
        if plot:
            fig, ax = plt.subplots()
            ax.set_xscale('log', basex=2)
            ax.set_yscale('log', basey=10)

            plt.plot(result[0], result[1], 'm*-', ms = 10, mec = None)
            try :
                plt.plot(result[0], result[2], 'co--', ms = 8, mec = None)
            except IndexError:
                pass
            plt.xlabel('Number of data points $n$')
            plt.ylabel('Time of execution $\mu$secs$')
            plt.legend(methods, 'upper left')
            plt.show()

Now, lets compare the performance of 2 methods- Naive and KDTree implementations.

In [243]: perf_test(methods = ['Tree', 'Naive'], n_list = [2 ** i for i in range(4, 14)], plot = True)

Took 2.00009 msecs with Tree method for 16 datapoints
Took 1.99986 msecs with Tree method for 32 datapoints
Took 2.00009 msecs with Tree method for 64 datapoints
Took 3.00002 msecs with Tree method for 128 datapoints
Took 4.99988 msecs with Tree method for 256 datapoints
Took 11.0002 msecs with Tree method for 512 datapoints
Took 20.9999 msecs with Tree method for 1024 datapoints
Took 48.0001 msecs with Tree method for 2048 datapoints
Took 106 msecs with Tree method for 4096 datapoints
Took 179 msecs with Tree method for 8192 datapoints
Took 3.00002 msecs with Naive method for 16 datapoints
Took 3.00002 msecs with Naive method for 32 datapoints
Took 6.00004 msecs with Naive method for 64 datapoints
Took 13 msecs with Naive method for 128 datapoints
Took 30.9999 msecs with Naive method for 256 datapoints
Took 82.9999 msecs with Naive method for 512 datapoints
Took 249 msecs with Naive method for 1024 datapoints
Took 834 msecs with Naive method for 2048 datapoints
Took 3734 msecs with Naive method for 4096 datapoints
Took 15796 msecs with Naive method for 8192 datapoints

We see that KDTree outperforms Naive method for large $n$, but it may not do well for small number of datasets. In my PC, I cannot run Naive method beyond $2^{13}$ datapoints, or else I receive MemoryError. So, let's evaluate the performance of KDtrees up to 1 Million datapoints.

In [244]: perf_test(methods = ['Tree'], n_list = [2 ** i for i in range(4, 21)], plot = True)

Took 2.00009 msecs with Tree method for 16 datapoints
Took 2.00009 msecs with Tree method for 32 datapoints
Took 1.99986 msecs with Tree method for 64 datapoints
Took 3.00002 msecs with Tree method for 128 datapoints
Took 6.00004 msecs with Tree method for 256 datapoints
Took 9.00006 msecs with Tree method for 512 datapoints
Took 20 msecs with Tree method for 1024 datapoints
Took 50 msecs with Tree method for 2048 datapoints
Took 108 msecs with Tree method for 4096 datapoints
Took 194 msecs with Tree method for 8192 datapoints
Took 396 msecs with Tree method for 16384 datapoints
Took 837 msecs with Tree method for 32768 datapoints
Took 1741 msecs with Tree method for 65536 datapoints
Took 3596 msecs with Tree method for 131072 datapoints
We can see, algorithm is scaling well with data-set size $n$. If we analyse the complexity of algorithm, its linearithmin, i.e. $\Theta(n \log n)$.

In [228]: