Robust, Scalable Anomaly Detection for Large Collections of Images

Michael S Kim
Cloud Analytics
Booz Allen Hamilton
Email: kim_michael3(at)bah.com

***Abstract*—A novel robust anomaly detection algorithm is applied to an image dataset using Apache Pig, Jython and GNU Octave. Each image in the set is transformed into a feature vector that represents color, edges, and texture numerically. Data is streamed using Pig through standard and user defined GNU Octave functions for feature transformation. Once the image set is transformed into the feature space, the dataset matrix (where the rows are distinct images, and the columns are features) is input into an original anomaly detection algorithm written by the author. This unsupervised outlier detection method scores outliers in linear time. The method is linear in the number of outliers but still suffers from the curse of dimensionality (in the feature space). The top scoring images are considered anomalies. Two experiments are conducted. The first experiment tests if top scoring images coincide with images which are marked as outliers in a prior image selection step. The second examines the scalability of the implementation in Pig using a larger data set. The results are analyzed quantitatively and qualitatively.**

I. Introduction

The motivation of this paper comes from the movement to leverage large scale, distributed computing for practical objectives such as remote sensing. Those in the intelligence community may want to monitor activity in areas such as cities where there is much variation. Cameras have become ubiquitous and thus petabyte scale databases of images are not uncommon. Hence this paper solves the challenging engineering problem of scalable anomaly detection.

There are three central concepts which are explored in this paper. One is calculating features for images based upon color, edges and texture via statistical methods. This is fairly straightforward because it applies conventional methods such as the calculation of moments. The second concept is the original anomaly detection algorithm. The third concept addresses the engineering problem of implementing anomaly detection on large data sets via Pig streaming through GNU Octave. The last two concepts of the anomaly detection algorithm and its Pig implementation are the major contributions of this paper to the big data community.

The original algorithm scores outliers in R^m. The method is an empirical cumulative distribution function (ECDF) approach that relies upon resampling methods. Each point j in a data set S has an ECDF describing the distribution of Euclidean distances from point j to other points in set S. A point k is an outlier if its average Kolmogorov Smirnov (KS) statistic across all other points in S is large. Since the KS statistic is bounded between 0 and 1, the average KS statistic is bounded between 0 and 1. We call the average KS statistic, the KSE statistic since it is based upon resampling (Efron) from the original dataset.

The algorithm is implemented in Apache Pig using Jython User Defined Functions (UDFs). Pig passes a stream of image file locations to GNU Octave which reads each image as a three dimensional array of doubles. Then feature transformation is performed in Pig streaming through GNU Octave functions. Each image is transformed into a feature vector which becomes a point for the anomaly detection algorithm to score. Anomaly detection is performed in Pig where several Jython UDFs perform various numerical methods to compute the outlier score for each image. The scores are output by Pig into a CSV.

II. Experiment 1

The first experiment tests the anomaly detection algorithm on a small data set of images where the ground truth is assigned for each image. This small scale experiment was performed in GNU Octave (without Pig and Jython).

A. Data

The input data is a collection of 20 images of broccoli. Of these 20 images, the last two images (indexed as 19 and 20) are anomalies. Image 19 is a cartoon picture of broccoli. Image 20 is an artificial cutout of broccoli. The rest of the images marked 1-18 are all photographs of the actual vegetable representation of broccoli.

Each of these images are transformed into a feature vector in GNU Octave. There was an imperfect effort to avoid multicollinearity in this process of feature selection. Since many possible features exist, the focus was on color, edges and texture. The color features are determined by an algorithm which examines each pixel of an image. Pixels are classified as red, green, blue or other. For a pixel, if a channel X is greater in intensity (by 5 points) than the other channels, the pixel is classified as channel X. The decimal proportion of red, green and blue pixels are used as features. Edge based features used built in GNU Octave functions such as Laplacian of Gaussian, thresholding, and median filtering [1] to obtain a black and white matrix of an image's edges. The decimal proportion of black pixels in this matrix was calculated as a feature. Moran's I is another feature which was extracted from the black and white (binary) edge matrix [2]. Moran's I calculation used a weight

[1]Median filtering does not work with Pig Streaming and hence is not included in the Pig based system build

[2]Because Moran's I is computationally expensive a downsampled version of the image matrix is used as the input.

matrix based upon the Rook's case of neighbors with weight 1 (and all others at 0)[3]. For the equation given, n is the number of pixels for a given image, w_{ij} is taken from the weight matrix, x_i and x_j are pixel values, and $\bar{x}$ is the average pixel value for a given image.

$$I_{morans} = \frac{n}{\sum_i^n \sum_j^n w_{ij}} \frac{\sum_i^n \sum_j^n w_{ij}(x_i - \bar{x})(x_j - \bar{x})}{\sum_i^n (x_i - \bar{x})^2} \tag{1}$$

Moran's I is calculated from the edge image and the raw grayscale image. It measures spatial autocorrelation for the grayscale image and the edges of the image. Hence we obtain Moran's I in these two cases separately. The last features are extracted from the grayscale image via statistical methods. One can summarize the empirical CDF of the pixel values of the grayscale image. This is done in GNU Octave by calculating the minimum, first quartile, median, third quartile, maximum, mean, standard deviation, skewness, and kurtosis of the vector of image pixel values. Using the above methods, each image in the dataset is transformed to a real vector of the features described in this section [4].

B. Methods

After each image is transformed to the feature space, we are left with a matrix where the rows are distinct images and the columns are features. This matrix can be input into the anomaly detection algorithm which is described here.

C. Algorithm

Suppose we have n data points in R^m [5]. Call this set of points S. Our objective is to give each data point in S an outlier score where a higher score implies higher confidence that the point is an anomaly.

[3]For a given pixel at ij, the weight of 1 is given to the pixels one up, one down, one left, and one right (and the pixel in question). The rest of the pixels are given a weight of 0.

[4]The appendix contains details of the exact implementation of feature transformation including specific parameter values used

[5]These are n images in a m-dimensional feature space.

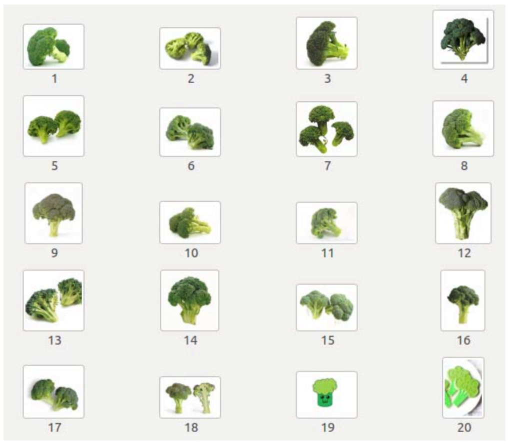

Fig. 1. The collection of images of broccoli. The last two (19 and 20) are marked as anomalies.

Any given point j in S has an ECDF [6] $F_{p_j}(x)$ describing the distribution of Euclidean distances from point j to other points in set S. We can compute the Kolmogorov Smirnov statistic between point j and other points in S.

$$KS(p_j, p_i) = \sup_x |F_{p_j}(x) - F_{p_i}(x)| \tag{2}$$

We can take the average of these Kolmogorov Smirnov test statistics to compute the KSE test statistic that is the outlier score.

$$KSE(p_j) = \frac{1}{n-1} \sum_{\substack{i=1 \\ i \neq j}}^{n} KS(p_j, p_i) \tag{3}$$

A point is an outlier if its average Kolmogorov Smirnov (KS) statistic across other points in S is large. Since the KS statistic is bounded between 0 and 1, the average KS statistic is bounded between 0 and 1. We call the average KS statistic, the KSE statistic.

The algorithm above for scoring n points in R^m can be reduced in complexity via sampling. The original algorithm is $O(m*n^3)$, but it can be reduced to $O(C*m*n)$ where C is a possibly large sampling constant that does not depend on n or m. Each point has an ECDF of Euclidean distances to other points. Calculating each Euclidean distance in R^m takes m steps. We can estimate the ECDF instead of taking the actual ECDF. This can be done by sampling $C_1 < n-1$ other points. This also applies when calculating the average KS test statistic. Instead of taking an average KS stat using all $n-1$ points, we sample $C_2 < n-1$ via random sampling with replacement [7]. All sampling done here is random sampling with replacement (also known as a bootstrap sample), but later other methods can also be used. This results in a time complexity of $O(C_1 * C_2 * m * n) = O(C * m * n)$. As n (data points) grows very large, we can adjust C_1 and C_2 at the cost of precision. When n is small a $O(m*n^3)$ algorithm can be used without loss of any precision.

D. Experiment

The experiment is to see if the highest scored points correspond to the images marked as outliers [8]. If one marks the highest scored points as outliers, what is the decimal proportion of points which have to be marked as outliers such that all outliers are accounted for? Ideally if there are α outliers where $0 < \alpha < 1$ then taking the α top scoring points should account for all outliers in the dataset. This would mean the outlier detection algorithm has a 0% false positive and false negative rate. In general, one would likely have to take more than the top α scoring points to account for all the outliers in the data.

[6]Empirical cumulative distribution function

[7]We can also sample without replacement as done in the Matlab implementation using the randperm function. Apache Pig's sample function is a probabilistic algorithm for random sampling (without replacement).

[8]The unsampled version of the algorithm with $O(m*n^3)$ time complexity is used for experiment 1

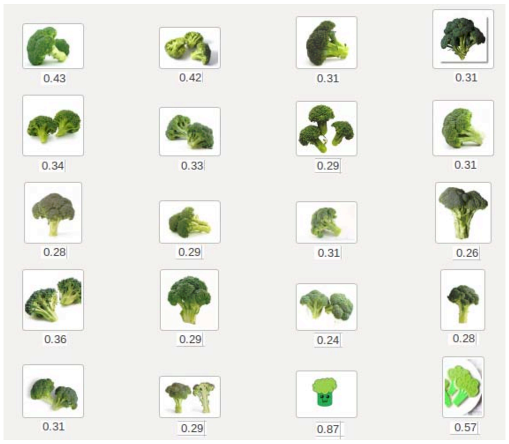

Fig. 2. The collection of images of broccoli. The anomaly detection algorithm's scores are listed for each picture.

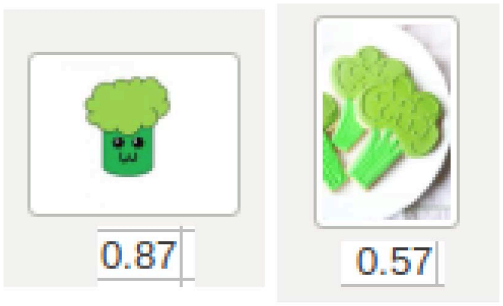

Fig. 3. The highest scoring images (19 and 20) which are the outliers.

E. Experimental Results

The experimental results show the outlier detection algorithm correctly identifies outliers with a 0% false positive and false negative rate in this dataset. However, the power of these results is small given the limited amount of data used. Future work must consider much larger datasets.

Image 19 which is a cartoon drawing of broccoli had the highest KSE score of 0.87. The next highest score belonged to image 20 with a KSE score of 0.57. Since images 19 and 20 are the previously marked outliers, the anomaly detection algorithm correctly identified the anomalies.

Analysis of the KSE scores shows that similar images may have similar scores. Images 3, 7 and 9 share the same score of 0.31 and are remarkably similar. This trend continues with the images with moderately low KSE scores of 0.28. Hence it is likely the same techniques used for anomaly detection could be applied to clustering.

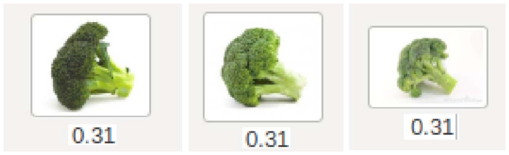

Fig. 4. The images 3, 7 and 9 with the score of 0.31.

III. Experiment 2

The second experiment is an implementation of the anomaly detection algorithm on a Linux Ubuntu 12.04 system of Apache Pig 0.11.1, Jython 2.5.3 and GNU Octave 3.2.4 [9]. In this experiment, we examine the runtime and qualitatively examine the highest scoring and lowest scoring images.

A. System

Since the experiment was conducted through Pig local on a single quad-core machine, the system architecture is fairly simple. All nodes in the system should be running Linux Ubuntu 12.04 and Apache Pig 0.11.1. Apache Pig requires certain dependencies such as Hadoop to be installed. GNU Octave 3.2.4 should be installed with the 'image' package version 2.0.0 or higher. The GNU Octave *.m script file should have certain permissions set [10].

The Pig script reads in file names of where images are located in HDFS (or locally). GNU Octave reads in each image and creates a feature vector through standard image processing techniques. Pig with Jython UDFs then scores each image using the KSE statistic technique.

B. Data

The data was obtained through Google image search of the term "elephant" with safe search turned on. The first 1000 images are used as input into the anomaly detection system. The images were obtained using Firefox with the DownThemAll plugin.

C. Runtime

The system took approximately 2 hours to score 1000 images. The Pig script's KSE statistics two sampling parameters were set at 60 each. GNU Octave computed Moran's I using downsampled to 60 by 60 pixel images. Given all scripts were run locally, there is potential for runtime improvement given access to a large Hadoop cluster. Sampling parameters can also be adjusted to trade precision for runtime speed.

D. Highest and Lowest Scoring Images

The images with the 6 lowest KSE scores are photographs of elephants in the wild. The majority of the pictures in the dataset involve photographs of elephants in the wild. Hence we would expect non-outliers to be photographs of elephants. The lowest KSE scores are between 0.263 and 0.264.

The images with the 6 highest KSE scores are not photographs of elephants in the wild. The top scoring images or anomalies are drawings of abstract elephants or not elephants at all. Such images are not common within the dataset. The KSE scores of the top outliers range from 0.772 to 0.793.

[9] All tests are conducted on an Alienware X51 Desktop Windows 8 running Ubuntu 12.04 via Virtual Box 4.2.12.

[10] This can be done in terminal with the command "chmod 755 octavefilename.m"

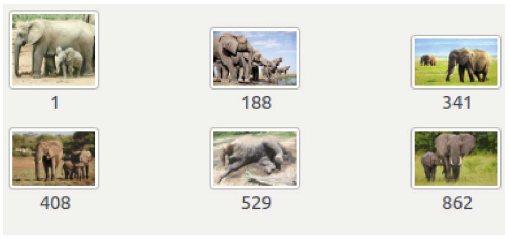

Fig. 5. The common images with KSE scores between 0.263 and 0.265.

IV. Conclusion

This paper has shown the potential of anomaly detection using image transformation, resampling, and empirical CDF based techniques. Some work has already been conducted on examining other datasets with moderate success. However, examination of larger amounts of data will require more computational resources and time. Since the anomaly detection algorithm works in Pig, scaling the system out is not a problem.

The anomaly detection algorithm can be modified in terms of metric used and resampling scheme. In some cases of very high dimensional data, distance measures (such as cosine similarity) that are not proper metrics might be used. The resampling scheme is also subject to change. Subsampling, jackknife and the standard bootstrap are all possible choices that have different asymptotic properties depending on the underlying data.

Future engineering work could involve different ways to integrate mathematical languages such as GNU Octave or R with existing big data tools such as Pig. This would allow cloud based signal processing of images, video and sound. Potential extensions of the algorithm can include clustering of the KSE scores in linear time.

V. Appendix

This section includes the source code used in this paper. The white space in the code has been modified to fit into the IEEE two column paper format. This can affect the code for languages where white space is part of the syntax, such as Jython.

A. GNU Octave Code

```
#!/usr/bin/octave -qf
#requires "chmod 755 filename.m"
#start image manipulation

while (!feof(stdin))
```

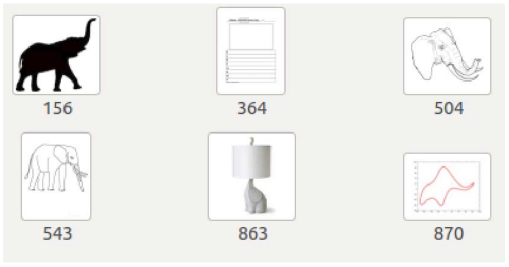

Fig. 6. The anomalies with the highest KSE scores within the elephant dataset.

```
#functions
addpath("/usr/share/octave/packages/3.2/image-1.0.14");
addpath("/usr/share/octave/packages/3.2/");
pkg load all;

function [tmpRed, tmpGreen, tmpBlue] = calc_color(inputIM)
  [M, N, Z] = size(inputIM);
  tmpRed = 0; tmpGreen=0; tmpBlue=0;
  tmpTotal =0;
  for k = 1:M
    for l = 1:N
      tmpTotal = tmpTotal +1;
      if (inputIM(k,l,1) > inputIM(k,l,2)+5 &&
      inputIM(k,l,1) > inputIM(k,l,3) +5)
        tmpRed = tmpRed + 1;
      elseif (inputIM(k,l,2) > inputIM(k,l,1)+5 &&
      inputIM(k,l,2) > inputIM(k,l,3)+5)
        tmpGreen = tmpGreen + 1;
      elseif (inputIM(k,l,3) > inputIM(k,l,2)+5 &&
      inputIM(k,l,3) > inputIM(k,l,1)+5)
        tmpBlue = tmpBlue + 1;
      endif
    end
  end
  tmpRed = tmpRed/tmpTotal;
  tmpGreen = tmpGreen/tmpTotal;
  tmpBlue = tmpBlue/tmpTotal;
endfunction

function morans = moransi(inputIM)
  moranIM = double(imresize(inputIM,[30,30]));
  tmpMean = mean(moranIM(:));
  iNum = 0; iDenom = 0; totalN=0; sumW =0;
  [M, N] = size(moranIM);
  for k = 2:(M-1)
    for l = 2:(N-1)
     iDenom = iDenom + (moranIM(k,l)-tmpMean)^2;
     totalN = totalN+1;
     for o = 2:(M-1)
       for p = 2:(N-1)
         # rook's case, 4 cases
         if ((k==o && l == p+1) || (k==o && l == p -1) ||
         (l==p && k == o+1) || (l==p && k == o-1) || (k==o && l==p ))
           iNum = iNum + (moranIM(k,l) - tmpMean)
           *(moranIM(o,p)-tmpMean);
           sumW = sumW + 1;
         endif
       endfor
     endfor
    endfor
  endfor
  morans = totalN * iNum / (sumW * iDenom);
endfunction

zz = fgets(stdin);
inpic = zz(1:(length(zz)-1));
inputIM = imread(inpic);

%grayscale features
if isrgb(inputIM) == 1
[rr,gg,bb] = calc_color(inputIM);
inputIM = double(rgb2gray(inputIM));
else
inputIM = double(inputIM(:,:,1));
rr=0.33; gg=0.33; bb=0.33;
end

statsIM = statistics(inputIM(:));
moransGray = moransi(inputIM);

%calculate edges features
GI= imfilter(inputIM,  fspecial("gaussian"));
LI= imfilter(GI,  fspecial("laplacian"));
edgeparam = prctile(double(LI(:)),90);
[M, N] = size(LI);

bwLI = LI;
tmpB=0; tmpW=0;
for k = 1:M
    for l = 1:N
     if LI(k,l) > edgeparam
      bwLI(k,l) = 0;
      tmpB= tmpB+1;
     else
      bwLI(k,l)=255;
      tmpW = tmpW +1;
     end
    end
end

%edge based features
featureBlack = tmpB / (tmpB+tmpW);
moransEdge = moransi(bwLI);

tmpFeatures = [rr, gg, bb, featureBlack, statsIM',
moransGray, moransEdge];
tmpFS = mat2str(tmpFeatures);
tmpLen = length(tmpFS)-1;
tmpFS = tmpFS(2:tmpLen);
printf(strcat(inpic,',',tmpFS, '\n'));
endwhile
```

B. Jython UDFs

```
#!/usr/bin/python
# Filename - ks_average.py
@outputSchema("mean_ks:double")
def ks_average(data):
 sumx = 0
 length = 0
 data1 = []
 for tup in data:
  sumx = sumx + tup[(len(tup) - 1)]
  length = length + 1
 return float(sumx)/float(length)

 #!/usr/bin/python
# Filename - ks_test.py
@outputSchema("ks_test_stat:double")
def ks_test(data_1, data_2):
 data1 = []
 data2 = []
 for tup in data_1:
  data1.append(tup[(len(tup) - 1)])
 for tup in data_2:
  data2.append(tup[(len(tup) - 1)])

 n1 = len(data1)
 n2 = len(data2)
 n = n1 + n2
 data1.sort()
 data2.sort()
 data_all = data1 + data2
 data_all.sort()
 cdf1 = []
 cdf2 = []

 def dist1(ind):
  dist = 1e100000
  if x-data1[ind] >= 0:
   dist = x - data1[ind]
  return dist

 def dist2(ind):
  dist = 1e100000
  if x-data2[ind] >= 0:
   dist = x - data2[ind]
  return dist

 for x in data_all:
  if x >= max(data1):
   cdf1.append(1.0)
  elif x < min(data1):
   cdf1.append(0.0)
  else:
   cdf1.append((1.0 + min(reversed(range(n1)), key=dist1))/(1.0*n1))

  if x >= max(data2):
   cdf2.append(1.0)
  elif x <  min(data2):
   cdf2.append(0.0)
  else:
   cdf2.append((1.0 + min(reversed(range(n2)), key=dist2))/(1.0*n2))

 d=0
 for x in range(n):
  if abs(cdf1[x] - cdf2[x]) > d:
   d = abs(cdf1[x] - cdf2[x])
 return d
```

C. Apache Pig Script

```
-- pig -x local anomalyPics.pig
REGISTER './UDF/ks_average.py' USING jython AS statfuncs;
REGISTER './UDF/ks_test.py' USING jython AS statfuncs;

filenames = LOAD './data/picLocations.txt'
USING PigStorage('\t') AS (x1: chararray);

DEFINE CMD `/home/mikekim/Desktop/pigscripts/UDF/transformImages.m`;
points = STREAM filenames THROUGH CMD;
STORE points INTO './tmp_output' USING PigStorage(',');
EXEC;
-----

points = LOAD './tmp_output/part*' USING PigStorage(',') AS
 (x1: chararray, x2: double, x3: double, x4: double,
 x5: double, x6: double, x7: double,
x8: double, x9: double, x10: double,
x11: double, x12: double, x13: double,
x14: double, x15: double, x16: double);

points = FOREACH points {
GENERATE $0 AS id,
 $1 AS x1, $2 AS x2, $3 AS x3, $4 AS x4,
 $5 AS x5, $6 AS x6, $7 AS x7, $8 AS x8,
 $9 AS x9, $10 AS x10, $11 AS x11, $12 AS x12,
 $13 AS x13, $14 AS x14, $15 AS x15;};

points_sample = SAMPLE points 0.9;
points_sample = LIMIT points_sample 60;

dists = FOREACH (CROSS points, points_sample) {
sq_dist = (points::x1 - points_sample::x1)*
(points::x1 - points_sample::x1) +
 (points::x2 - points_sample::x2)*
 (points::x2 - points_sample::x2) +
 (points::x3 - points_sample::x3)*
 (points::x3 - points_sample::x3) +
 (points::x4 - points_sample::x4)*
 (points::x4 - points_sample::x4) +
 (points::x5 - points_sample::x5)*
 (points::x5 - points_sample::x5) +
 (points::x6 - points_sample::x6)*
 (points::x6 - points_sample::x6) +
 (points::x7 - points_sample::x7)*
 (points::x7 - points_sample::x7) +
 (points::x8 - points_sample::x8)*
 (points::x8 - points_sample::x8) +
 (points::x9 - points_sample::x9)*
 (points::x9 - points_sample::x9) +
 (points::x10 - points_sample::x10)*
 (points::x10 - points_sample::x10) +
 (points::x11 - points_sample::x11)*
 (points::x11 - points_sample::x11) +
 (points::x12 - points_sample::x12)*
 (points::x12 - points_sample::x12) +
 (points::x13 - points_sample::x13)*
 (points::x13 - points_sample::x13) +
 (points::x14 - points_sample::x14)*
 (points::x14 - points_sample::x14) +
 (points::x15 - points_sample::x15)*
 (points::x15 - points_sample::x15);
GENERATE points::id, SQRT(sq_dist) as dist;
};

dists = GROUP dists BY id;
dists_sample = SAMPLE dists 0.9;
dists_sample = LIMIT dists_sample 60;
kse_stats = CROSS dists, dists_sample;

-- for each of the pairs above compute kse test stat,
-- then group by id and take the average stat
kse_stats = FOREACH kse_stats
GENERATE $0 AS id, statfuncs.ks_test($1, $3) AS kse;
by_kse = GROUP kse_stats BY id;
by_kse = FOREACH by_kse
GENERATE $0 AS id, statfuncs.ks_average($1) AS avg_kse;
final_points = ORDER by_kse BY $1 DESC;
STORE final_points INTO './output' USING PigStorage(',');
```

REFERENCES

[1] B. Efron, *Bootstrap Methods: Another Look at the Jackknife*. The Annals of Statistics, 1979.

[2] M.S. Kim, *Anomaly Detection*. www.mathworks.com/matlabcentral/fileexchange/39593-anomaly-detection, 2012.

[3] F.R. Hampel, et al., *Robust statistics: the approach based on influence functions*. Wiley, 2011.

[4] R. C. Gonzales and P. Wintz, *Digital Image Processing*. Addison-Wesley, 1977.

[5] Octave community, *GNU Octave*. www.gnu.org/software/octave/, 2013.

[6] Apache Pig community, *Apache Pit 0.11.1*. http://pig.apache.org/, 2013.