Anomalous Data Detection

Gary Steelman, University of Texas at Dallas

CS 6375 Machine Learning

gxs112030@utd.edu | gary.steelman42@gmail.com

Introduction

As computers grow more powerful, the volume of data generated also grows. Sometimes parts of the data should be considered *anomalous* – data that doesn't match the expected observation. It is nearly impossible to manually check all generated data for anomalies because of the time it would take. Machine learning can be used to assist with anomalous data detection.

Anomalous data detection has many real world applications such as fraud detection, hardware manufacturing quality control, sensor networks, and robotics. If an anomaly is detected, a bank can look into whether someone's identity has been stolen. If a manufactured part doesn't withstand testing, it can be pulled from the assembly line and remade, perhaps saving lives. A sensor network can realize when a node is going to die, a robot can know when not to overreact to stimulus.

To detect anomalous data I build multiple *classifiers* – data modelers which receive an observed data and report whether the observed data should be considered anomalous or not.

Presented in this paper are three distinct algorithms for learning three distinct anomalous data classifiers. The first using univariate Gaussians, the second using a multivariate Gaussian, and the third using k-means classifiers. The first two algorithms are widely used while the third is of my own invention.

Results show that by using a relatively naive model I can achieve relatively high accuracy.

Formal Definition and Algorithms

The problem of anomalous data detection for the purposes of this paper is precisely defined as:

Given an observed data vector *x* with n attributes, $a_1, \ldots, a_n \in \mathbb{R}^n$ determine if *x* is an anomalous observation (true) or a normal observation (false).

Presented are three algorithms to accomplish the task – each created with a different philosophy in mind. I present the pseudo code for each algorithm, followed by a short discussion for each. I discuss the design, philosophy, pros, and cons of each algorithm.

It is useful to note that when reading data in from a file, the models all may attempt to pre-partition the data into a normal data set *D* and an anomalous data set *AD.*

There are multiple variables and data symbols common to each algorithm. They are defined below:

```
n = number of attributes
D = Normal data matrix (m x n)
D[i] = ith row of D
D[i][j] = jth column of D
AD = Anomalous data matrix (t x n)
AD[i] = ith row of AD
AD[i][j] = jth column of AD
x = observed data vector (1 x n)
UG = Array of n UnivariateGaussian
MG = 1 MultivariateGaussian
eps = classification barrier for Gauss
models
KM = Array of n KMeans for D
KMA = Array of n KMeans for AD
```

Model 1 – Univariate Gaussians

("Machine Learning | Coursera")

Train(D, AD):

```
// Train the Gaussians
for j = 0 to n
  g = new UnivariateGaussian
  mu = compute mean of D[i][j]
  v = compute variance D[i][j]
  g.populate(mu, v)
  UG[j] = g

// Calculate epsilon
DMinP = INF
for i = 0 to m
  p(D[i]) = 1.0
  For j = 0 to n
    p(D[i]) *= UG[j].sample(D[i][j])
  if p(D[i]) < DMinP
    DMinP = p(D[i])

ADMaxP = 0
for i = 0 to t
  p(AD[i]) = 1.0
  for j = 0 to n
    p(AD[i]) *= UG[j].sample(AD[i][j])
  if p(AD[i]) > ADMaxP
    ADMaxP = p(AD[i])

eps = (DMinP + ADMaxP) / (m + t)
```

Classify(x):

```
// Calculate p(x)
p = 1.0
for j = 0 to n
  p *= UG[j].sample(x[j])

// Compare to classification barrier
if p < eps
  return true
else
  return false
```

The algorithm trains *n* univariate Gaussian models, one for each attribute. It then calculates the maximum likelihood estimate mean,

$$\mu_{MLE} = \frac{1}{n}\sum_{i=0}^{n} x_i$$

and then the maximum likelihood estimate variance,

$$\sigma^2_{MLE} = \frac{1}{n}\sum_{i=0}^{n} (x_i - \mu_{MLE})^2$$

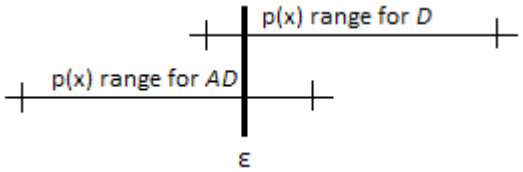
for each attribute in *D* and sends those two values to the corresponding Gaussian. Each univariate Gaussian can then later be sampled to obtain $p(x|\mu,\sigma^2)$, the probability of seeing a value *x* given the means and variance learned earlier,

$$P(x|\mu_{MLE}, \sigma^2_{MLE}) = \frac{\exp\left[\frac{-(x-\mu_{MLE})^2}{2\sigma^2_{MLE}}\right]}{\sigma_{MLE}\sqrt{2\pi}}$$

Then the algorithm attempts to learn a good ε, the classification boundary,

$$\epsilon = \frac{\min[p(x)\forall x \in D] + \max[p(x)\forall x \in AD]}{m+t}$$

which is the minimum probability for a single point in *D* plus the maximum probability for a single point in *AD* over the sum of the size of *D* and *AD*. This is the classification boundary for which observed p(x) > ε is considered normal and p(x) < ε is anomalous. Graphically it looks like,



To classify an observed data *x*, the algorithm feeds each attribute in *x* to the corresponding univariate Gaussian and p(Y|x) is calculated as,

$$p(Y|x) \propto \alpha \prod_{j=0}^{n} p(x_j|\mu_{MLE}, \sigma^2_{MLE})$$

where $p(x_j)$ is the value returned from the j^{th} univariate Gaussian.

Essentially this multiplies the probabilities of observing the each value in *x* and uses that as the total probability for *x*.

Pros:
+ Polynomial complexity yields fast training.
+ Polynomial complexity yields fast classification.
+ If the distribution of values for an attribute is close to normal, a univariate Gaussian learns it very closely.

Cons:
- If the distribution of values for an attribute is not close to normal, a univariate Gaussian introduces high bias.
- Any covariance or correlation of attributes is not captured.
- Learns solely on *D*, ignoring *AD*.
- A poorly chosen ε annihilates accuracy.

Model 2 – Multivariate Gaussian

("Machine Learning | Coursera")

("Calculation of Matrix Inverse in C/C++")

("6.5.4.1. Mean Vector and Covariance Matrix")

("Lecture Slides (Machine Learning, CS 6375)")

Train(D, AD):

```
// Train the Gaussian
for j = 0 to n
  g = new MultivariateGaussian
  g.populate(D)
  MG = g

// Calculate epsilon
DMinP = INF
for i = 0 to m
  p(D[i]) = MG.sample(D[i][j])
  if p(D[i]) < DMinP
    DMinP = p(D[i])

ADMaxP = 0
for i = 0 to t
  p(AD[i]) = MG.sample(AD[i][j])
  if p(AD[i]) > ADMaxP
    ADMaxP = p(AD[i])

eps = (DMinP + ADMaxP) / (m + t)
```

Classify(x):

```
// Calculate p(x)
p = MG.sample(x)

// Compare to classification barrier
if p < eps
  return true
else
```

```
return false
```

The algorithm *looks* more simple than the univariate Gaussian model presented. It trains a single multivariate Gaussian of *n* parameters on *D*. The multivariate Gaussian computes the maximum likelihood estimate mean vector,

$$\vec{\mu}_{MLE} = \frac{1}{n}\sum_{i=0}^{m}\vec{x}_i$$

where x_i are the rows of *D* . The *nxn* covariance matrix, whose entries are computed one at a time,

$$\Sigma_{i,j} = \frac{\sum_{k=0}^{m}(D_{k,i}-\mu_{MLE\,i})(D_{k,j}-\mu_{MLE\,j})}{n-1}$$

$$where\ 0 \le i \le n, 0 \le j \le n$$

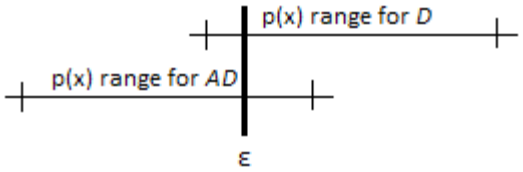
The multivariate Gaussian can later be sampled to obtain P(x|μ,Σ), the probability of seeing a data vector *x* given the means and covariance matrix learned earlier,

$$P(\vec{x}|\vec{\mu}_{MLE},\Sigma) = \frac{\exp\left[-\frac{1}{2}(\vec{x}-\vec{\mu}_{MLE})^T\Sigma^{-1}(\vec{x}-\vec{\mu}_{MLE})\right]}{2\pi^{\frac{n}{2}}\sqrt{|\Sigma|}}$$

Then the algorithm attempts to learn a good ε, the classification boundary the same way the univariate Gaussian model does,

$$\epsilon = \frac{\min[p(x)\forall x \in D] + \max[p(x)\forall x \in AD]}{m+t}$$

which is the minimum probability for a single point in *D* plus the maximum probability for a single point in *AD* over the sum of the size of *D* and *AD*. This is the classification boundary for which observed p(x) > ε is considered normal and p(x) < ε is anomalous. Graphically it looks like,



To classify an observed data *x*, the algorithm passes the vector *x* to the multivariate Gaussian and checks the returned value against ε.

Essentially this relies on the strengths of a multivariate Gaussian to encapsulate all the information from the univariate Gaussians model plus inter-attribute covariance.

Pros:

+ Polynomial complexity yields fast training.
+ Polynomial complexity yields fast classification.
+ Captures data covariance and correlation in the covariance matrix.
+ If the distribution for the data is close to normal, a multivariate Gaussian learns it very closely.

Cons:

- If the distribution for the data is not close to normal, a multivariate Gaussian introduces high bias.
- Must compute the inverse and determinant of the covariance matrix. Sometimes this isn't possible and the model will fail completely.
- Added complexity over univariate Gaussians model makes it slightly slower.
- Learns solely on *D*, ignoring *AD*.
- A poorly chosen ε annihilates accuracy.

Model 3 – Array of K-Means

("Lecture Slides (Machine Learning, CS 6375)")

```
Train(D, AD):
// Train the Normal KMeans array
for j = 0 to n
  k = new Kmeans with 1 cluster
  k.populate(D[i][j])
  KM[j] = k

// Train the Anomalous KMeans array
if t > 0
  for j = 0 to n
    k = new Kmeans with 1 cluster
    k.populate(AD[i][j])
    KMA[j] = k
```

```
Classify(x):

// Calculate attribute-wise distances
// to cluster centers
// Anomalous data present
if t > 0
  for j = 0 to n
    distN = dist(KM[j].center, x[j])
    distAN = dist(KMA[j].center, x[j])
    if distAN < distN
      return true
  return false

// No anomalous data present
else
  for j = 0 to n
    distN = dist(KM[j].center, x[j])
    if distN > KM[j].maxdist
      return true
  return false
```

The algorithm is a departure from the pattern of the first two. The idea is to compute a cluster center for

each attribute of *D* and *AD* and then do attribute-wise distance calculations from *x* to those cluster centers. If an attribute in *x* is closer to the *AD* center, then *x* is flagged as anomalous.

The algorithm computes a KMeans of one cluster for each attribute in *D* and in *AD*. Given only one cluster for each attribute, this effectively yields the maximum likelihood estimate mean for each attribute,

$$center = \mu_{MLE} = \frac{1}{n}\sum_{i=0}^{n} x_i$$

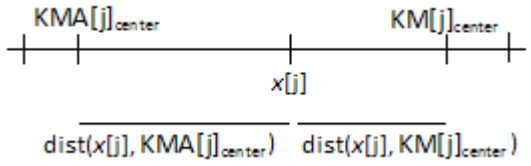
As each classifier computes its center it also records the maximum distance from its center to any point in the cluster,

$$\text{dist}(x, y) = \sqrt{x^2 + y^2}$$
$$maxdist = \max(\text{dist}(x, x_{center})) \; \forall x \in cluster$$

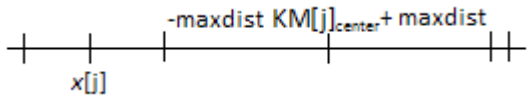
which simply uses 1-D Euclidean distance.

To classify an observed data *x*, the KMeans model has two options: 1) If there is any known anomalous data present in the training set or 2) if no known anomalous data is present in the training set.

For option 1) the algorithm loops over each attribute in *x*. It computes the distance from an attribute to the corresponding KMeans model's cluster centers. It then checks if the distance to the anomalous center is closer than the distance to the normal center. If it is, *x* is classified as anomalous. Graphically this looks like,



For option 2) the algorithm loops over each attribute in *x*. It computes the distance from an attribute to the corresponding KMeans model's normal cluster center. It then checks if the distance to the cluster center is farther than the cluster center's farthest observed point when it was trained. If it is, *x* is classified as anomalous. Graphically this looks like,



Essentially, this approach creates a range of 2*maxdist around each cluster center and any points not in the range are classified as anomalous.

Pros:
+ Easy to visualize and understand.
+ Polynomial complexity yields fast training.
+ Polynomial complexity yields fast classification.
+ Learns using *D* and *AD*.

Cons:
- Depending on what represents an anomalous value, distance measures can quickly overflow.
- If the normal data isn't distributed as a circle around each cluster center, classifications will be inaccurate.
- Attributes with extremely high variance destroy accuracy (see Results for details).

Experimental Evaluation

The data used for training and testing for all three algorithms is the same: real-world sensor network data sampled every 30 seconds from Carnegie-Mellon University's campus. The data is composed of 6-tuples in the form:
x = *<time, nodeid, temperature humidity, light, voltage>*
The data set has over 2.3 million entries. ("10-701 Machine Learning Fall 2007")

Hypothesis

All three models will achieve high accuracy on the test data set.

Loading

("Cfloat (float.h) - C++ Reference")

Each algorithm loads the input data in the exact same way, which changes depending on the model mode. Examples which do not contain anomalous values are placed into the *normal set, D* (for data). Examples which contain anomalous values are placed into the *anomalous set, AD* (for anomalous data). Each algorithm considers all examples in *D* to be examples of "ok", "normal", or "expected" observations. The AnomalyDetector mode has multiple values: SUPERVISED, SEMI_SUPERVISED, and UNSUPERVISED.

If the mode is SUPERVISED or SEMI_SUPERVISED when loading begins, the detector attempts to partition the input data *D* and *AD.* It look for values like NaN or INF in the input data, replaces these values with DBL_MAX (the C++ maximum value representable by a double type) and places any tuple containing these values into *AD*. After loading completes, if *AD* empty, the detector falls back to UNSUPERVISED mode.

If the mode is UNSUPERVISED when loading begins, no

partitioning for the input data occurs and all examples are placed into *D*.

In either loading case, the following property holds:

$$input\ data = D \cup AD$$

D and *AD* are stored as matrices of n columns, one for each attribute, and m and t rows respectively, one for each example,

$$[D]_{m\,x\,n} \text{ and } [AD]_{t\,x\,n}$$

Training

D and AD are partitioned further into two subsets each,

$$D = D_{train} \cup D_{test}$$
$$AD = AD_{train} \cup AD_{test}$$

where the training sets contain a random X% of the data and the test sets contain the remaining Y% of the data.

The algorithms are then run for *k*-fold cross validation. Each fold partitions D_{train} and AD_{train} further into two subsets, a training set for the iteration containing a random W% of the data and a cross validation set containing the remaining Z% of the data,

$$D_{train} = D_{train,i} \cup D_{cv,i}$$
$$AD_{train} = AD_{train,i} \cup AD_{cv,i}$$

For the i^{th} fold, the algorithm uses the i^{th} training partition to learn and then tests the accuracy of the classifier learned using the i^{th} cross validation set,

$$accuracy_{cv,i} = \frac{D_{correct\ classifies} + AD_{correct\ classifies}}{|D_{cross\ validation,i}| + |AD_{cross\ validation,i}|}$$

The model which produces the highest accuracy on the cross validation sets is the model kept by the detector,

$$classifier_{best} = \underset{accuracy_{cv}}{\text{argmax}} \{classifiers\}$$

Testing

Each algorithm was tested in exactly the same way. All examples in D_{test} and AD_{test}, stored from the training step were classified. These two test sets were fed to the induced $classifier_{best}$ and the accuracy was computed as with the training cross validation,

$$accuracy_{test} = \frac{D_{correct\ classifies} + AD_{correct\ classifies}}{|D| + |AD|}$$

Runs for the algorithms were completed multiple times and the resulting accuracies were averaged and are are reported in the Results section.

Results

Discussion

The results obtained are using a 2.3 million example data set. The train/test data split is a 70%/30% split. For cross validation, the train/cv split is 85%/15%.

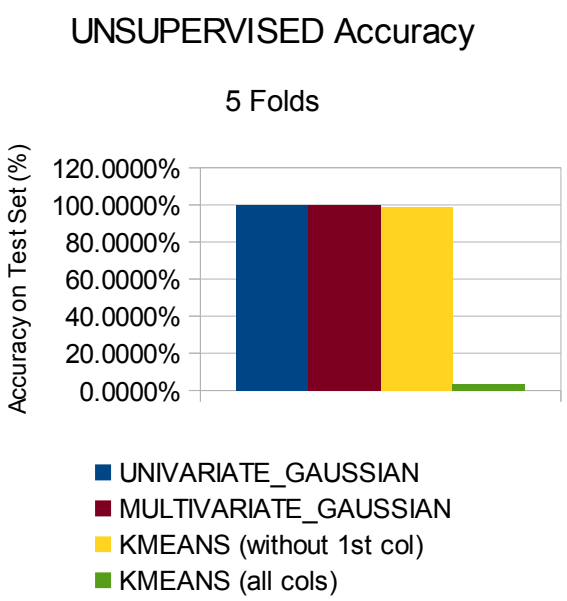
An interesting phenomena was discovered during testing for the Array of KMeans algorithm: including the first column of the data (the time stamp) caused accuracy to decrease significantly. Further investigation showed this was because of how the data was distributed: with an extremely high variance. Attributes containing a high variance (which also have a fairly random distribution of anomalous values) cause the accuracy of the algorithm to tank.

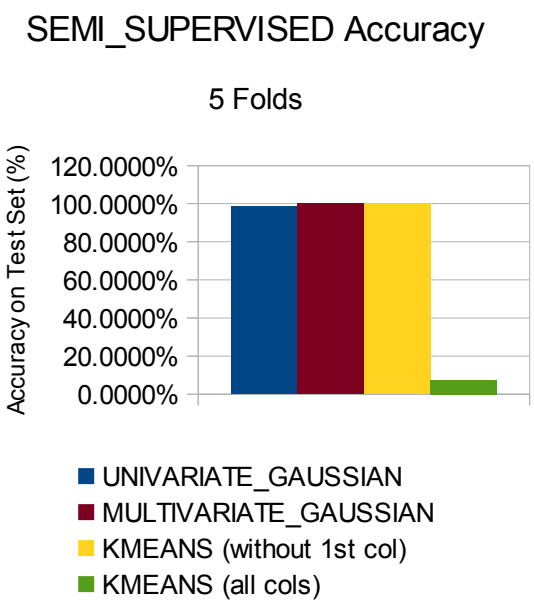
Additionally with the Array of KMeans algorithm and SEMI_SUPERVISED mode, the value which replaces NaN or INF in the input data must be carefully chosen. In this instance, DBL_MAX was used. To achieve highest accuracy, the chosen value should be a value as different from the real values in the data as possible. Ie, if the data values range from 0.0 to 500.0, DBL_MAX or -DBL_MAX are a great choice. If the values range from 10^{250} to DBL_MAX, then choosing DBL_MAX will result in poor accuracy. Instead, use -DBL_MAX.

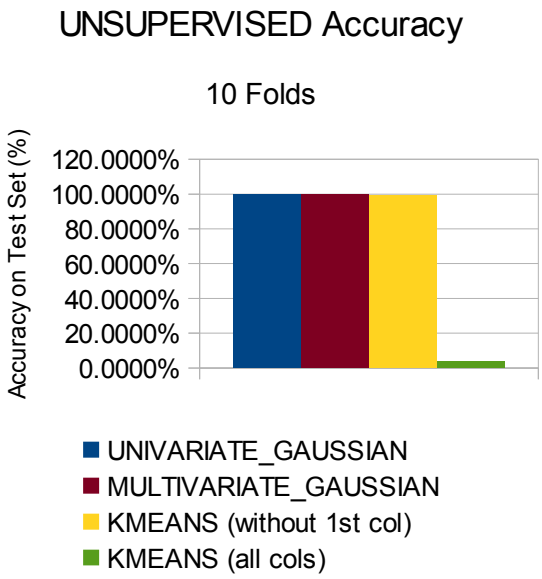
Figures

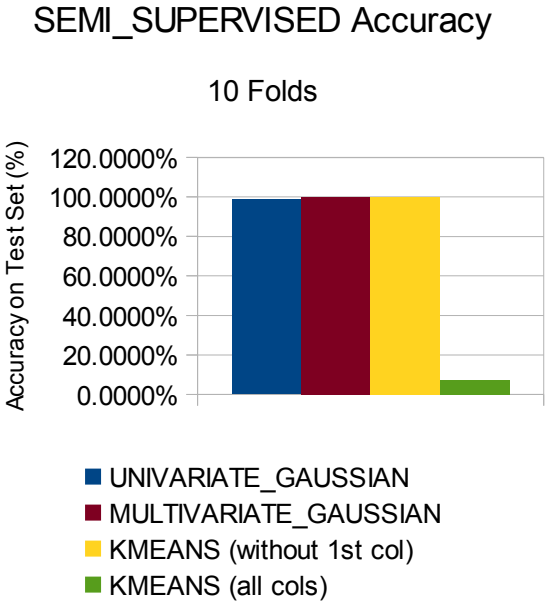
Shown below are four charts. The first two show the average accuracy over three runs for SEMI_SUPERVISED mode for five and ten folds, respectively.

The second two show the average accuracy over three runs for UNSUPERVISED mode for five and ten folds, respectively.









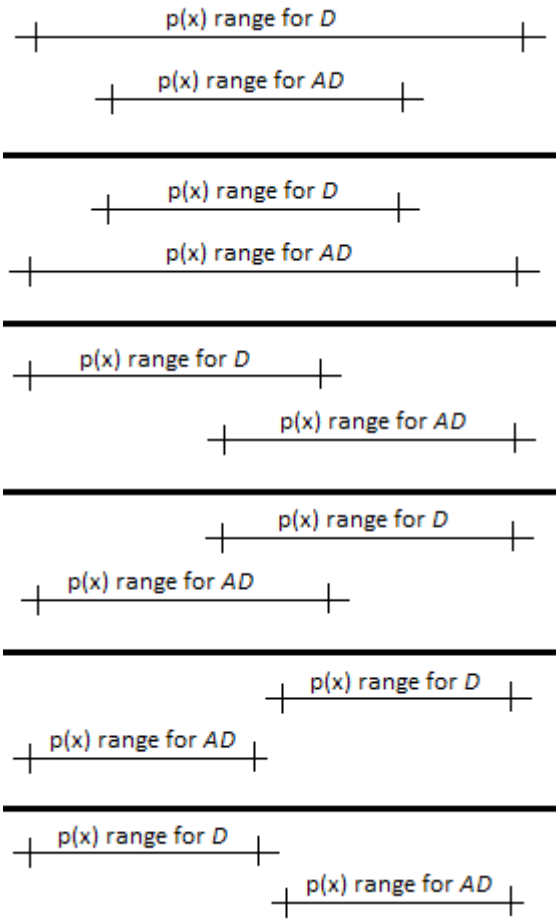
For more information see runs.(ods|pdf).

Related and Future Work

Discussed below are a few ways to improve the presented algorithms to cover some of the weaknesses of each.

Epsilon Calculation for Gaussian Models

The ε calculation for the Gaussian models is naïve and assumes either the 4th or 5th case shown below. The intervals for range of probabilities for the $D_{train,\ i}$ and $AD_{train,\ i}$ could be different. There are six cases for the intervals of the probabilities,



Which, clearly, given the presented ε calculation will fail miserably in any but the 4th and 5th cases. Additionally, the p(x) vs ε check for Gaussian model classification could be flipped to allow for the 3rd or 6th case to work.

EM Replacement of KMeans

The Array of KMeans model could be improved by instead using soft assignment for clustering, as with an EM classifier. This would simplify the initial data partitioning for the algorithm. The data would not need to be partitioned into *D* and *AD*, but could be left in *D.* The examples could be probabilistically assigned as normal or anomalous rather than by distance calculation.

Use of Pre-Classified Anomalous Data

Each of the algorithms presented has a weakness in its inability to maximize use of any pre-classified anomalous data in *AD*. The univariate Gaussians model could learn the distribution of attributes for *AD* too use it to calculate p(x). The multivariate Gaussian model could learn an additional multivariate Gaussian for *AD* and again, use that information to calculate p(x). The KMeans model already uses *AD.*

Conclusions

I have constructed three different anomalous data detection algorithms and tested them against a real-world data set from Carnegie Mellon University. The results indicate that with a little tweaking the algorithms presented can induce learners that achieve high accuracy. Training of classifiers is not fast enough for real-time application, but classification is fast enough for real-time application. The hypothesis holds and all classifiers achieved high accuracy if the data adhered to the constraints discussed.

Bibliography

"10-701 Machine Learning Fall 2007." Web. 26 Apr. 2012.

"6.5.4.1. Mean Vector and Covariance Matrix." Web. 27 Apr. 2012.

"Calculation of Matrix Inverse in C/C++." *Jason Yu-Tseh Chi's Notes*. Web. 27 Apr. 2012.

"Cfloat (float.h) - C++ Reference." Web. 27 Apr. 2012.

"CS 391L: Machine Learning." Web. 26 Apr. 2012.

"Lecture Slides (Machine Learning, CS 6375)." Web. 27 Apr. 2012.

“Machine Learning | Coursera.” Web. 27 Apr. 2012.