Chapter 1 Introduction to Data Mining

1.1 What is Data Mining (DM)?

There are many different definitions of Data Mining. There are some commonly accepted definitions, such as

1. Non-trivial extraction of implicit, previously unknown and potentially useful information from data.
2. Exploration and analysis, by automatic or semi-automatic means, of large quantities of data in order to discover meaningful patterns.

Data mining is a process of discovering **useful information** and **patterns** hidden in a huge dataset. Note that looking up phone number in phone directory or query a web search engine is **information retrieval** rather than data mining.

Data Mining is originated from Statistics, Machine learning, Artificial Intelligence, Pattern recognition and database systems.

1.2 Major Tasks

There are several major tasks in DM:

1. Classification
2. Regression
3. Cluster Analysis
4. Deviation Detection
5. Association Rule Discovery
6. Sequential Pattern Discovery

1.3 Challenges

1. Scalability. Due to the advances of technology, huge dataset with sizes of gigabytes (2^30), terabytes (2^40), or even petabytes (2^50) are common.
2. Dimensionality. It is now common to encounter dataset with hundreds or thousands of attributes or variables.
3. Complex Data. Data can be of different types: Nominal, Ordinal, Interval, Ratio and other non-traditional types: Hyperlink and XML document, DNA sequence, Graph etc.
4. Data Quality. Dataset may contain many outliers and missing values.
5. Data Fusion. Sometimes we may have to combine dataset from different sources.
6. Privacy Preservation. On one hand, DM aims at discover hidden information and pattern about customers and on the other hand it may violate customers' privacy.

1.4 Data Matrix and variable type

Traditional dataset is arranged in a matrix form with each row as record (or observation) and each column as attribute (or variable). This matrix is called a data matrix and is represented as follow:

$$\begin{bmatrix} x_{11} & \cdots & x_{1p} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{np} \end{bmatrix}$$

n is the number of records and p is number of attributes. In practice, n is much larger than p. These p variables can be of different types. Usually the variables can be classified into the following types:

Type	Description	Examples	Operations
Nominal	Just label or different name to distinguish one object from another.	Zip code, ID, Gender.	= or not =
Ordinal	The values provide the ordering of objects.	Opinion, Grades.	< or >
Interval	Unit of measurement, but the origin is arbitrary.	Celsius or Fahrenheit, Calendar dates.	+ or -
Ratio	Unit of measurement and the origin is not arbitrary.	Temperature in Kelvin, Length, Counts, Age, Income.	+, -, *, /

Nominal and ordinal variable are discrete while interval and ratio variable are continuous. Sometimes, an interval or ratio variable can be coded as an ordinal variable, e.g. Income, Age etc; while ordinal variable can be treated as continuous variable.

1.5 Missing values

In real dataset, we often encounter missing values as well. The easiest way to handle missing values is to throw away all the incomplete observations. R has a build-in function `complete.cases()` to select all the complete cases. However, using only the complete cases will throw away lots of information and more seriously may lead to a biased sub-sample. However, this may leads to a biased sample if the values are **non-randomly missing**. Let us illustrate this by a simple simulation using R. First we simulate 5000 random numbers from N(0,1) and transform into a 500x10 data matrix as follow:

```
> set.seed(12345)                  # set random seed
> d<-matrix(rnorm(5000),ncol=10)   # generate d
```

Now we replace all the values in d which are less than -2 to NA (missing value) and save it to x1. Similarly we replace values which are greater than 2 to NA and save it to x2. Finally we replace values either less than -2 or greater than 2 to NA and save it to x3.

```
> x1<-replace(d,d<(-2),NA)     # left truncation
> x2<-replace(d,d>2,NA)        # right truncation
> x3<-replace(x1,x1>2,NA)      # truncated at both ends
```

Now we select all the complete cases in x1, x2 and x3 and save them to c1, c2 and c3.

```
> c1<-x1[complete.cases(x1),] # or using c1<-na.omit(x1)
> c2<-x2[complete.cases(x2),] # c2<-na.omit(x2)
> c3<-x3[complete.cases(x3),] # c3<-na.omit(x3)
```

Finally, we compare the column mean and standard deviation of d, c1, c2, and c3.

```
> apply(d,2,mean)              # compute column mean
 [1]  0.082460774  0.009935544 -0.043519610 -0.017265351 -0.049974047
 [6] -0.001067202 -0.030677799  0.063169727 -0.038143052  0.027272271

> apply(c1,2,mean)
 [1]  0.1233467858  0.0638723710  0.0084738799  0.0521666953  0.0009470797
 [6] -0.0039870233  0.0457505594  0.1031269644  0.0191648779  0.0883821627

> apply(c2,2,mean)
 [1]  0.03824293 -0.08345927 -0.11063462 -0.04921670 -0.07007709 -0.02017169
 [7] -0.06523789 -0.02188608 -0.10117069 -0.08671383

> apply(c3,2,mean)
 [1]  0.08885456 -0.01085131 -0.04744800  0.01507342 -0.02685057 -0.02054034
 [7]  0.00570698  0.01449568 -0.05521309 -0.04347896
```

From the above results, *c1* (left truncation) over-estimated the true mean, *c2* (right truncation) under-estimated the true mean while *c3* (truncated at both ends) somewhat gives an unbiased estimates.

```
> apply(d,2,sd)
 [1] 0.9901489 1.0069571 1.0065296 1.0110109 0.9504727 0.9633377 0.9366497
 [8] 0.9668011 1.0032361 1.0477286
> apply(c1,2,sd)
 [1] 0.9400771 0.9433430 0.9496533 0.9511094 0.8835571 0.9228504 0.8661841
 [8] 0.9278996 0.9408468 1.0094124
> apply(c2,2,sd)
 [1] 0.9197499 0.9362285 0.9887511 1.0036806 0.9044965 0.9078869 0.8931388
 [8] 0.8902894 0.9506731 0.9610994
> apply(c3,2,sd)
 [1] 0.8388386 0.8464405 0.9247468 0.9612472 0.8385220 0.8720126 0.8208492
 [8] 0.8409841 0.8547181 0.8959722
```

For the column *sd, c1* and *c2* under-estimated the true sd, while *c3* under-estimated the true sd even further.

1.6 Dissimilarities (or Distances) between objects

Measure of dissimilarities or distances between objects are important since many DM techniques, such as **k-means clustering** and **nearest neighbor classification** are based on this measure. It is important to define distance between two observations. Since there may be many different types of variables in our data set, the distance or dissimilarity between objects (observations) should be defined differently according to their variable type. Suppose that we have p variables in our data set and the object *i* and *j* denoted by

object	*i*	x_{i1}	$\cdots$	x_{ip}
object	*j*	x_{j1}	$\cdots$	x_{jp}

Continuous variables

If all the variables are continuous, we can use either one of the following as the distance measure:

Euclidean : $d(i,j) = \sqrt{(x_{i1} - x_{j1})^2 + \ldots + (x_{ip} - x_{jp})^2} = \sqrt{\sum_{k=1}^{p} (x_{ik} - x_{jk})^2}$

City-Block(Manhattan) : $d(i,j) = | x_{i1} - x_{j1} | + \ldots | x_{ip} - x_{jp} | = \sum_{k=1}^{p} | x_{ik} - x_{jk} |$

Minkowski: $d(i,j) = [| x_{i1} - x_{j1} |^q + \ldots | x_{ip} - x_{jp} |^q]^{1/q} = [\sum_{k=1}^{p} | x_{ik} - x_{jk} |^q]^{1/q}$

Ordinal variables

If all the variables are ordinal, we can rescale the observation r_k to [0,1] by

$z_{ik} = (r_{ik} - 1)/(M_k - 1)$ *and* $z_{jk} = (r_{jk} - 1)/(M_k - 1)$.

where variable *k* has ranks *1,2,…,* M_k. Then we can treat them like continuous variables.

Binary variables

If all the variables are binary, we can use either one of the following measures:

- Simple matching coefficient:
  d(i,j) = (r+s)/(q+r+s+t) = (r+s)/p
- Jaccard coefficient:
  d(i,j) = (r+s)/(q+r+s)

		Object *j*	
		1	0
Object *i*	1	*q*	*r*
	0	*s*	*t*

Note that the Jaccard coefficient ignores the counts for both *i* and *j* are zero. This is used for the situation when *t* is much bigger than *q+r+s*, (e.g. the transaction records of items in supermarket).

Nominal variables

If all the variables are nominal, we can use

d(i,j) = (p-m)/p where m is the number of matches among these *p* variables.

Scaling of data

In practice, we often encounter different types of variables in the same dataset. Another issue is that the range of the variables may differ a lot. Using the original scale may put more weights on the variables with large range. An unified approach is to re-scale them into the [0,1] interval before performing data analysis. Assume that there are p variables in the dataset.

Define the distance between the *i*-th and *j*-th observation as $d(i,j) = \sum_{k=1}^{p} d_{ij}^{(k)}$.

If variable *k* is continuous or ordinal, then $d_{ij}^{(k)} = |x_{ik} - x_{jk}| / (\max_u x_{uk} - \min_u x_{uk})$.

If variable *k* is binary or nominal with *m* classes (say $x_k = 1$ or 2 …, or m), we first create *m* dummy binary variables as follows:

x_k	w_1	w_2	⋯	w_m
1	1	0	⋯	0
2	0	1	⋯	0
⋮	⋮	⋮		⋮
m	0	0	⋯	1

, then $d_{ij}^{(k)} = \sum_{h=1}^{m} \frac{1}{2n_h} |w_{ih} - w_{jh}|$, where $w_h = \begin{cases} 1 & \text{if } x_k = h \\ 0 & \text{otherwise} \end{cases}$.

where n_h is the frequency of w_h in category *h* in the dataset.

The following functions *scale.con()* and *scale.dum()* will scale the continuous/ordinal and binary/nominal variables accordingly, $z_{ik} = (x_{ik} - \min_u x_{uk}) / (\max_u x_{uk} - \min_u x_{uk})$.

```
# function to scale continuous or ordinal variables to [0,1]

scale.con<-function(d) {
  n<-dim(d)[1]            # row dim of d
  p<-dim(d)[2]            # column dim of d
  cmin<-apply(d,2,min)    # column min of d
  cmax<-apply(d,2,max)    # column max of d
  range<-cmax-cmin        # column range
  cmin<-matrix(cmin,nr=n,nc=p,byrow=T)   # change cmin to a nxp matrix
  range<-matrix(range,nr=n,nc=p,byrow=T)  # change range to a nxp matrix

  (d-cmin)/range # transform d
}
```

```
# function to convert categorical variable v with k levels to k dummy variables
cat2dum<-function(v) {
  v<-factor(v)              # change v to factor
  lab<-levels(v)            # get value in v
  k<-length(lab)            # get no. levels in v
  x<-outer(v,lab,"==")+0    # create matrix x with k columns
  x                         # output
}
# function to scale nominal variables to dummy variables
scale.dum<-function(d) {
  n<-dim(d)[1]          # row dim. of d
  p<-dim(d)[2]          # col. dim. of d
  x<-NULL               # initalize x
  for (i in 1:p) {
    v<-d[,i]            # get the i-th col in d
    z<-cat2dum(v)       # convert into z, matrix dummy var.
    x<-cbind(x,z)       # column-binding of z
  }
  k<-dim(x)[2]                          # col. dim of x
  nh<-apply(x,2,sum)                    # compute frequency of each category
  nk<-matrix(nh,nrow=n,ncol=k,byrow=T)  # create nxk matrix with each row is nh
  x/(2*nk)                              # output
}
```

To illustrate how scale.dum() works, let us create a data matrix *d* with 4 rows and 3 columns, the first column is categorical variable with 4 levels while columns 2 and 3 are binary.

```
> d<-matrix(c(2,0,1,3,1,0,1,0,0,4,0,1),nrow=4,byrow=T)    # create data matrix d
> d                                                       # display d
     [,1] [,2] [,3]
[1,]    2    0    1
[2,]    3    1    0
[3,]    1    0    0
[4,]    4    0    1
> cat2dum(d[,1])                # apply cat2dum to 1st col. of d
     [,1] [,2] [,3] [,4]
[1,]    0    1    0    0
[2,]    0    0    1    0
[3,]    1    0    0    0
[4,]    0    0    0    1
> cat2dum(d[,2])                # apply cat2dum to 2nd col. of d
     [,1] [,2]
[1,]    1    0
[2,]    0    1
[3,]    1    0
[4,]    1    0
> cat2dum(d[,3])                # apply cat2dum to 3rd col. of d
     [,1] [,2]
[1,]    0    1
[2,]    1    0
[3,]    1    0
[4,]    0    1

> d1<-scale.dum(d)              # apply scale.dum on d
> d1
     [,1] [,2] [,3] [,4]      [,5] [,6]  [,7] [,8]
[1,]  0.0  0.5  0.0  0.0 0.1666667  0.0  0.00 0.25
[2,]  0.0  0.0  0.5  0.0 0.0000000  0.5  0.25 0.00
[3,]  0.5  0.0  0.0  0.0 0.1666667  0.0  0.25 0.00
[4,]  0.0  0.0  0.0  0.5 0.1666667  0.0  0.00 0.25
```

Note that the frequency n_k in computing each column in *d1* are: *1,1,1,1,3,1,2,2*. The first 4 columns in *d1* is for *d[,1]*; next 2 columns is for *d[,2]* and the last 2 columns is for *d[,3]*.

From *d1*, we can compute the distances: $d_{12} = 0.5 + 0.5 + 0.1667 + 0.5 + 0.25 + 0.25 = 2.1667$.
Similarly, $d_{13} = 1/2 + 1/2 + 0 + 1/4 + 1/4 = 1.5$, $d_{14} = 1/2 + 1/2 + 0 + 0 = 1$,
$d_{23} = 1/2 + 1/2 + 1/6 + 1/2 = 1.6667$, $d_{24} = 1/2 + 1/2 + 1/6 + 1/2 + 1/4 + 1/4 = 2.1667$.
$d_{34} = 1/2 + 1/2 + 0 + 1/4 + 1/4 = 1.5$

These distances can be computed using the built-in function `dist()` as follow:

```
> dist(d1,diag=T,upper=T,method="manhattan")
         1        2        3        4
1 0.000000 2.166667 1.500000 1.000000
2 2.166667 0.000000 1.666667 2.166667
3 1.500000 1.666667 0.000000 1.500000
4 1.000000 2.166667 1.500000 0.000000
```

Note that all the above $d_{ij}^{(k)}$ have values in [0,1] and hence all the variables were treated equally when computing the dissimilarities or distances.

Standardization

If all the variables are continuous or ordinal, then another commonly used rescaling method is the standardized transformation: $z_{ij} = (x_{ij} - \bar{x}_j)/s_j$ for *i=1,…,n* and *j=1,…,p*. This standardized transformation will ensure the sample mean and sample s.d. of z_{ij} to be 0 and 1 respectively.

```
# function for standardize transformation
stand<-function(x) {
   n<-dim(x)[1]                    # row dim of x
   p<-dim(x)[2]                    # column dim of x
   m<-apply(x,2,mean)              # compute column mean
   s<-apply(x,2,sd)                # compute column sd
   m<-matrix(m,nr=n,nc=p,byrow=T)  # convert m into nxp matrix, each row is m
   s<-matrix(s,nr=n,nc=p,byrow=T)  # convert s into nxp matrix, each row is s
   (x-m)/s                         # output standardize score
}
```

1.7 Mahalanobis distance and Outlier detection

Dataset often contains outliers. Since many statistical techniques are sensitive to outliers, we usually detect and delete these outliers before applying the DM techniques. Usually we can standardize a variable by $z = (x - \bar{x})/s$, where $\bar{x}$ and s is the sample mean and standard deviation of *x* respectively; and consider the value |z|>3 as an outlier. However this can only detect part of the outliers as we only consider the marginal distribution of each variable. To detect multivariate outliers, we need more sophisticated methods. The **Mahalanobis distance** provides us a useful tool for detecting outliers. Let the random vectors $X_1, \ldots, X_n$ have a

p-variate normal distribution, that is, $X_1,\ldots,X_n$ i.i.d. $N_p(\mu,\Sigma)$. According to distribution theory, the Mahalanobis distance $D^2=(x-\bar{x})'S^{-1}(x-\bar{x})\sim\chi^2_p$, is distributed as a Chi-square distribution with p degrees of freedom. $\bar{x}$ is the px1 sample mean vector and *S* is the pxp sample covariance of random vector $X_1,\ldots,X_n$. This mean that if the observation with D^2 greater than a pre-assigned level (say 99-percentile) of a Chi-square distribution with p degrees of freedom, then this observation is an outlier.

Let us illustrate this by our first data set HMEQ. HMEQ contains baseline and loan performance information for 5960 recent home equity loans. The target (BAD) is a binary variable that indicates if an applicant eventually defaulted or was seriously delinquent. This adverse outcome occurred in 1189 cases (about 20%). For each applicant, 12 input variables were recorded:

Column	Name	Scale	Description
1	BAD	Binary	1=defaulted, 0=paid back
2	LOAN	Interval	Amount of loan
3	MORTDUE	Interval	Amount due on existing mortage
4	VALUE	Interval	Value of current property
5	REASON	Binary	HomeImp=Home improvement, DebtCon=debt consolidation
6	JOB	Nominal	Six occupational categories
7	YOJ	Interval	Years at present job
8	DEROG	Interval	Number of major derogatory reports
9	DELINQ	Interval	Number of delinquent trade lines
10	CLAGE	Interval	Age of oldest trade line in month
11	NINQ	Interval	Number of recent credit inquiries
12	CLNO	Interval	Number of trade lines
13	DEBTINC	Interval	Debt-to-income ratio

Let us read in this data set using R. For convenience, we select File -> change dir… in the menu to the folder that contains the data set.

```
> d<-read.csv("hmeq.csv",na.strings=" " )  # read in data set
> dim(d)                                   # display the dimension of d
[1] 5960   13
```

There are some missing values in the dataset and we assume that they are missing at random. We select only the complete cases from d and save them to *dc*.

```
> dc<-d[complete.cases(d),]   # select and save complete cases to dc
> dim(dc)                     # display dimension of dc
[1] 3364   13
> names(dc)                   # display the names in dc
 [1] "BAD"     "LOAN"    "MORTDUE" "VALUE"   "REASON"  "JOB"     "YOJ"
 [8] "DEROG"   "DELINQ"  "CLAGE"   "NINQ"    "CLNO"    "DEBTINC"
```

Since BAD is a target variable, we should calculated the distance separately for two groups: *BAD=0* and *BAD=1*. Therefore we create two matrices *d0* and *d1* as follow:

```
> d1<-dc[(dc$BAD==1),]     # select BAD=1
> dim(d1)                  # note that there is only 300 rows and 13 columns
 [1] 300  13
> d0<-dc[(dc$BAD==0),]     # select BAD=0
> dim(d0)                  # note that there is only 3064 rows and 13 columns
 [1] 3064   13
```

The following function mdist() in R to compute the Mahalanobis distance.

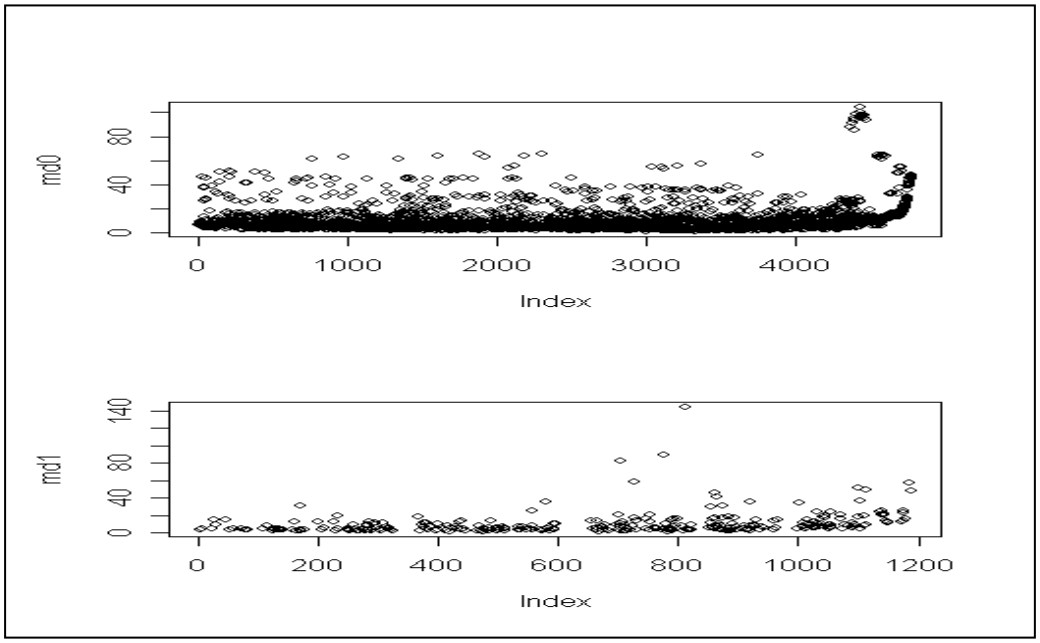
```
mdist<-function(x) {
   t<-as.matrix(x)         # transform x to a matrix
   m<-apply(t,2,mean)      # compute column mean
   s<-var(t)               # compute sample covariance matrix
   mahalanobis(t,m,s)      # using built-in mahalanobis function
}
```

We input these functions and apply the *mdist()* to *d0* and *d1*. Note that the 5th and 6th variables *REASON* and *JOB* are nominal and should be excluded in computing the Mahalanobis distance.

```
> source("mdist.r")           # load the file contains the mdist function
> md0<-mdist(d0[,-c(1,5,6)])  # compute distance and exclude columns 1,5,6
> md1<-mdist(d1[,-c(1,5,6)])  # and save results to md0 and md1
```

md0 and *md1* are vectors containing the Mahalanobis distance. If you are interested to see how these distances look like, you can plot these distances by the following commands:

```
> par(mfrow=c(2,1))       # set up a 2x1 multframe graphic
> plot(md0)               # plot md0
> plot(md1)               # plot md1
```



As seen from the plots, some values are very large and the corresponding observation vector is an outlier. Of course, we need to compute the 99-percentile of a Chi-square distribution with *p* d.f. (*p=10* in our example).

```
> c<-qchisq(0.99,df=10)
> c
[1] 23.20925
```

Therefore, any value in md0 or md1 which is greater than c is considered as outlier and should be excluded.

```
> x0<-d0[md0<c,]      # select observations with md0<c from d0
> x1<-d1[md1<c,]      # select observations with md1<c from d1
> dim(x0)             # x0 contains 2789 cases
[1] 2789   13
> dim(x1)             # x1 contains 278 cases
[1] 278  13
```

Finally we combine x0 and x1 row-wise and save the cleaned data set by

```
> x<-rbind(x0,x1)     # combine the matrices x0 and x1 row-wise
> dim(x)              # x have 3067 rows and 13 columns
[1] 3067   13
> write.table(x,file="hmeq1.csv",sep=",",row.names=F) # save x to hmeq1.csv
```

Note that the outliers may affect the calculation of $\bar{x}$ and S and hence the Mahalanbois distance D^2. There are some robust version of D^2, for example replacing mean by median and S by a robust version of S. However, the chi-square distribution result no longer true in this case and we have not going into the details.

Reference:

Chapters 1 and 2, Introduction to Data Mining by Tan, Steinbach and Kumar.