6. Neural Network: Bacteria Classification

Garth Mortensen

October 1, 2018

Image credit: Ousa Chea (https://unsplash.com/@cheaousa).

This analysis builds a Neural Network (NN) to classify bacteria cell DNA into two groups; *Flagged,* and *Not Flagged* for further study. The dataset consists of 14 columns and 1,217 observations. The target label is scaled 0 to 10, where 0 is not flagged for further study.

NNs are stacks of successive learning *layers* of increasingly meaningful represenations (Deep Learning with Python, 8). You can think of a deep network as a multistage information-distillation operation, where information goes through successive filters and comes out increasingly purified (9).

What makes NNs different is their improved performance on various problems, and simplification of problem solving since it automated the feature engineering step.

Overview

Perform an analysis on the cellDNA dataset using a Neural Network to predict which observations are worthy of further study, given 13 features.

Pre-Modeling

Neural Networks can require heavy computation, making them sometimes unfeasible for dealing with larger datasets. We'll measure how long this calculation takes, using a stopwatch method. This computer is running an i7-65000 with 16GB DDR4-2400 RAM. You can use CPU-Z (https://www.cpuid.com/softwares/cpu-z.html) to determine your RAM speed.

Start the stopwatch!

```
start_time <- Sys.time()
```

Load Required Packages

1. Install pacman using the following code.

```
#After experiencing my repository referencing the wrong mirror several times now, I keep this code available to avoid issue
s, especially when using knitr.
#local({r <- getOption("repos")
#       r["CRAN"] <- "http://cran.r-project.org"
#       options(repos=r)
#})

# Load required packages
if (!require("pacman")) install.packages("pacman")
```

```
## Loading required package: pacman
```

```
# If the above doesn't work, use the following. pacman is used for installing packages.
# #install.packages("pacman")
# library("pacman")
```

Load Data

The dataset we will be loading appears as:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	222	31.18919	40.34234	35.57909	8.883917	0.968325	-80.1137	222	1	16.81247	0.816176	0.578125	78.591	0
2	73	29.49315	271.3973	15.5172	6.40749	0.910764	76.04295	73	1	9.640876	0.858824	0.608333	39.217	0
3	256	58.81641	289.9414	37.22601	9.863895	0.964256	85.32474	256	1	18.05407	0.752941	0.562637	89.111	0
4	126	71.02381	477.4127	13.11298	12.79067	0.220351	63.52348	126	1	12.66602	0.881119	0.646154	43.832	0
5	225	90.80889	541.9467	44.46311	7.858879	0.984256	-52.875	225	1	16.92569	0.728155	0.252525	90.072	7
6	233	91.40773	279.6223	40.03947	9.256891	0.972908	80.28062	233	1	17.22396	0.732704	0.524775	88.623	0
7	288	109.7118	187.7882	42.20114	9.184247	0.976031	-0.63687	288	1	19.14923	0.842105	0.738462	87.192	0
8	260	102.3808	246.4115	35.06891	12.8204	0.930781	77.95412	263	-1	18.19457	0.638821	0.46595	99.788	0
9	231	106.0563	537.2078	39.62008	8.049715	0.979143	-48.1493	232	0	17.14988	0.868421	0.316872	80.958	0
10	100	100.94	167.05	19.943	8.591027	0.902457	-88.9441	100	1	11.28379	0.757576	0.584795	48.141	0
11	74	100.027	505.8919	14.15669	6.873529	0.874219	-82.0348	74	1	9.706685	0.91358	0.813187	31.534	0
12	102	103.2451	205.4608	13.39616	11.04129	0.56628	-88.6751	103	0	11.39607	0.842975	0.713287	43.435	0
13	424	118.1226	230.9835	66.00121	12.65948	0.981433	89.0279	425	0	23.23475	0.542894	0.422732	200.18	0
14	194	136.4742	566.4278	38.25739	7.172156	0.98227	-37.5776	194	1	15.7165	0.788618	0.281159	80.85	0
15	208	136.4231	165.4952	32.34916	9.873596	0.952282	67.69716	208	1	16.27372	0.8125	0.481481	70.892	0
16	195	133.159	213.3128	44.42961	6.459235	0.989376	88.61095	195	1	15.75696	0.691489	0.528455	97.059	4
17	250	145.428	200.828	52.46485	12.84421	0.96957	-67.2102	262	-3	17.84124	0.44405	0.227273	139.94	0
18	261	146.7011	251.8314	49.76097	8.921978	0.983795	84.63367	261	1	18.22952	0.659091	0.466905	102.856	9
19	363	193.2066	52.60606	54.37883	11.09028	0.978982	-33.9777	369	0	21.49851	0.610084	0.255814	141.272	0
20	187	176.8342	459.8235	35.19717	9.591662	0.962152	-85.0401	187	1	15.43035	0.619205	0.502688	82.486	0

Document Preview

```
cellDNA <- read.csv("C:/tmp/CellDNA.csv", header = TRUE)
```

Preview Data

Examine the data structure.

```
#Preview structure
str(cellDNA)
```

```
## 'data.frame':    1216 obs. of  14 variables:
##  $ X222         : int  73 256 126 225 233 288 260 231 100 74 ...
##  $ X31.18918919 : num  29.5 58.8 71 90.8 91.4 ...
##  $ X40.34234234 : num  271 290 477 542 280 ...
##  $ X35.57908668 : num  15.5 37.2 13.1 44.5 40 ...
##  $ X8.883916969 : num  6.41 9.86 12.79 7.86 9.26 ...
##  $ X0.968324558 : num  0.911 0.964 0.22 0.984 0.973 ...
##  $ X.80.11367302: num  76 85.3 63.5 -52.9 80.3 ...
##  $ X222.1       : int  73 256 126 225 233 288 263 232 100 74 ...
##  $ X1           : int  1 1 1 1 1 1 -1 0 1 1 ...
##  $ X16.81247093 : num  9.64 18.05 12.67 16.93 17.22 ...
##  $ X0.816176471 : num  0.859 0.753 0.881 0.728 0.733 ...
##  $ X0.578125    : num  0.608 0.563 0.646 0.253 0.525 ...
##  $ X78.591      : num  39.2 89.1 43.8 90.1 88.6 ...
##  $ X0           : int  0 0 0 7 0 0 0 0 0 0 ...
```

Examine the top 5 rows.

```
#Preview top 5 rows
head(cellDNA, n=5)
```

```
##   X222 X31.18918919 X40.34234234 X35.57908668 X8.883916969 X0.968324558
## 1   73     29.49315     271.3973     15.51720     6.407490    0.9107636
## 2  256     58.81641     289.9414     37.22601     9.863895    0.9642558
## 3  126     71.02381     477.4127     13.11298    12.790672    0.2203507
## 4  225     90.80889     541.9467     44.46311     7.858879    0.9842557
## 5  233     91.40773     279.6223     40.03947     9.256891    0.9729076
##   X.80.11367302 X222.1 X1 X16.81247093 X0.816176471 X0.578125 X78.591 X0
## 1      76.04295     73  1     9.640876    0.8588235 0.6083333  39.217  0
## 2      85.32474    256  1    18.054067    0.7529412 0.5626374  89.111  0
## 3      63.52348    126  1    12.666025    0.8811189 0.6461538  43.832  0
## 4     -52.87498    225  1    16.925688    0.7281553 0.2525253  90.072  7
## 5      80.28062    233  1    17.223960    0.7327044 0.5247748  88.623  0
```

Preprocessing

The dataset is raw and requires some tidying up. We'll give the dataset header names and drop the ID column, which carries no useful information.

```
#give col names
colnames(cellDNA) <- c("X1","X2", "X3", "X4","X5", "X6", "X7","X8", "X9", "X10","X11", "X12", "X13","Y")
```

We are going to give a binary classification to the Y target variable now, where anything other than 0 will equal to 1. The value 0 indicates a bacterium worthy of further study.

```
#ifelse(test, true, false)
cellDNA$Y <- ifelse(cellDNA$Y>0, 1, 0)
```

Because all measurements in our dataframe are in different units, we need to standardize. There are no dummy variables included.

```
#standardize all X variables
tempdf <- scale(cellDNA[,1:13])

#Bring Y back in
cellDNA = cbind(tempdf, cellDNA[,14])

#And reappoint column names
colnames(cellDNA) <- c("X1","X2", "X3", "X4","X5", "X6", "X7","X8", "X9", "X10","X11", "X12", "X13","Y")
```

Data Split

In machine learning, we feed data to the model, which then finds the statistical structure and builds the defining rules. We don't feed all the data to the model up front though. We split our dataframe into a training and test observations. We give the training data to the model to find an underlying statistical structure. Once the model has defined the rules, then we run them over the test data, and see how it's performed. Here, we split the data 70-30 into training and test data, using the caret package, which randomizes the split by default. By setting the random seed, we can all work to make our studies reproducable (https://en.wikipedia.org/wiki/Replication_crisis) for others.

```
pacman::p_load(caret)
library(caret)

# https://en.wikipedia.org/wiki/42_(number)#The_Hitchhiker's_Guide_to_the_Galaxy
set.seed(42)

#By default, createDataPartition does a stratified random split
#https://topepo.github.io/caret/data-splitting.html
trainIndex <- createDataPartition(
  #define y
  cellDNA[,14],
  #training set %
  p = 0.7,
  #results in a list (TRUE) or a matrix
  list = FALSE,
  #the number of partitions to create
  times = 1)

head(trainIndex)
```

```
##      Resample1
## [1,]         1
## [2,]         2
## [3,]         3
## [4,]         5
## [5,]         6
## [6,]         9
```

The above outputs an index. We now use this index to split the dataset.

```
training <- cellDNA[ trainIndex,]
testing  <- cellDNA[-trainIndex,]
```

The data has been split, and we can move onto the modeling phase.

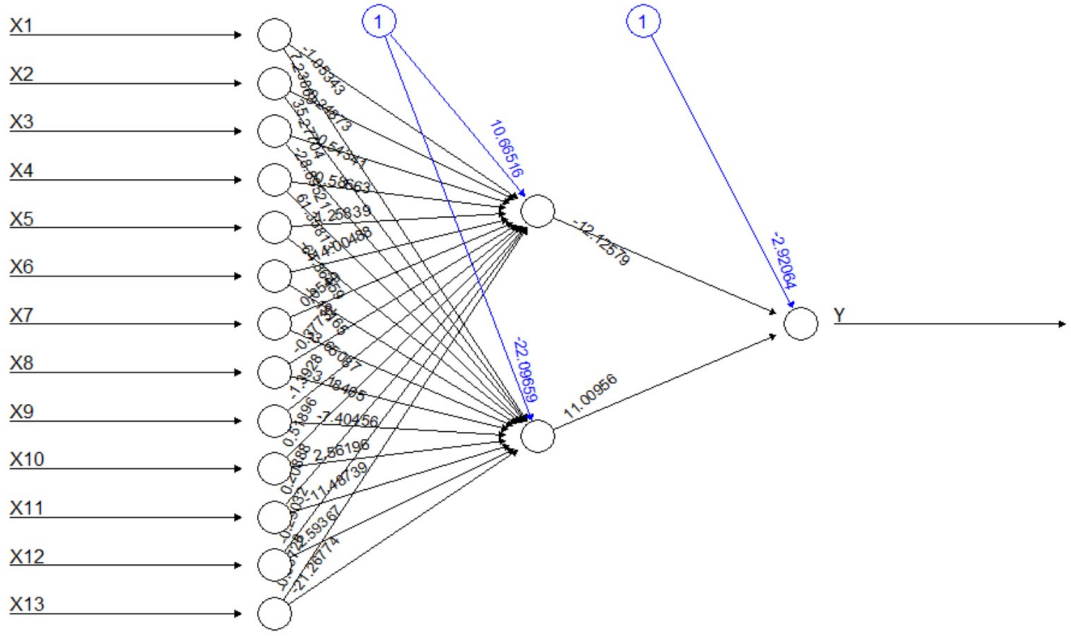
Modeling

We'll now use the neuralnet package and neuralnet algorithm to build a NN. Of note, this algorithm does not accept ~. notation (which is equivalent to "= all factors"). Instead, a verbose formula must be used.

```
pacman::p_load(neuralnet)
library(neuralnet)

#start watch fit
start_time_fit <- Sys.time()

fit <- neuralnet(Y ~ X1 + X2 + X3 + X4 + X5 + X6 + X7 + X8 + X9 + X10 + X11+ X12 + X13, data = training, hidden=c(2), linea
r.output=FALSE, threshold=0.01)

#stop watch fit
end_time_fit <- Sys.time()
total_time_fit <- end_time_fit - start_time_fit
total_time_fit = round(total_time_fit, digits = 2)
```

We decided on using a neural net configuration of 2, meaning two neurons in a single hidden layer. A linear relationship between the independent and dependent (target) variables is assumed to be false, and parameterized as such. Finally, if error changes during an iteration less than 1%, then optimization stops (threshold = 0.01)

```
#Print the results using the following
#fit$result.matrix
plot(fit)
```



Above we see a visual of our neural network. Black lines show connections with weights, which are calculated using back propogation. Blue lines display the bias term.

The above is difficult to read, especially when more inputs and nodes are used. To see the amount of error this fit does not explain, along with the determined weights between the inputs, hidden layers and output, use the following command.

```
fit$result.matrix
```

```
##                                    1
## error                     15.048634549382
## reached.threshold          0.009966803729
## steps                  12161.000000000000
## Intercept.to.1layhid1     10.665163051765
## X1.to.1layhid1            -1.053425671870
## X2.to.1layhid1             0.248727062944
## X3.to.1layhid1            -0.543409038326
## X4.to.1layhid1             0.586627893185
## X5.to.1layhid1             1.258388157790
## X6.to.1layhid1           -14.004879162366
## X7.to.1layhid1             0.054898011660
## X8.to.1layhid1            -0.377320959875
## X9.to.1layhid1            -1.392804652467
## X10.to.1layhid1            0.518957746345
## X11.to.1layhid1            0.208878332536
## X12.to.1layhid1           -0.290317236203
## X13.to.1layhid1           -0.667283580954
## Intercept.to.1layhid2    -22.096594512786
## X1.to.1layhid2             7.238625490171
## X2.to.1layhid2            35.277040126476
## X3.to.1layhid2           -28.895214847544
## X4.to.1layhid2            61.358126494177
## X5.to.1layhid2           -64.864587601185
## X6.to.1layhid2            -1.131647276166
## X7.to.1layhid2            -3.650865892410
## X8.to.1layhid2             3.184054588706
## X9.to.1layhid2            -7.404564745444
## X10.to.1layhid2            2.561963059542
## X11.to.1layhid2          -11.487389764364
## X12.to.1layhid2            2.593668794350
## X13.to.1layhid2          -21.267743203897
## Intercept.to.Y            -2.920643133165
## 1layhid.1.to.Y           -12.125792843611
## 1layhid.2.to.Y            11.009555289991
```

Validation

Training dataset

With our NN fitted, our next question is: how did it model the training data? We answer this by checking it's confusion matrix. Unlike Support Vector Machines, Decision Trees, Lasso Regression Models, etc, R's predict function will not work for NNs. Instead, we adopt a new method. For these steps, I closely followed this guide (http://www.michaeljgrogan.com/neural-network-modelling-neuralnet-r/).

```
# We subset to remove Y from the dataset
temp_training <- subset(training, select = c("X1","X2", "X3", "X4", "X5", "X6", "X7", "X8", "X9", "X10", "X11", "X12", "X1
3"))

# Compute function creates the prediction variable
# It computes the outputs of all neurons for specific arbitrary covariate vectors given a trained neural network.
# compute(x, covariate, rep = 1), where:
# x = an object of class nn.
# covariate = a dataframe or matrix containing the variables that had been used to train the neural network.
fit.results <- compute(fit, temp_training)
```

Having found our fit results, we can build a new dataframe containing the actual and predicted results. The new dataframe will consist of the actual Y column from testing, and the predicted column from fit.results. We title these two columns as actual and prediction, and print the first 10 rows.

```
results <- data.frame(actual = training[,14], prediction = fit.results$net.result)

# Display first 5 rows
results[1:10,]
```

```
##    actual         prediction
## 1       0 0.0000002924445271
## 2       0 0.0000008226925910
## 3       0 0.0000002920221245
## 4       0 0.0000031335178593
## 5       0 0.0000121695182583
## 6       0 0.0000002920860136
## 7       0 0.0000648270464336
## 8       0 0.0013632240282227
## 9       0 0.0764368102947272
## 10      0 0.0000010158546521
```

We can pivot these columns into a confusion matrix. Pivot seems like a fun way to express this. But before we do that, notice that our predicted values are not 0s and 1s. We need to round them to their closest integer.

Accuracy, Precision, Recall, FScore

```
rounded <- sapply(results, round, digits = 0)
rounded = data.frame(rounded) #redefine as df

# Is this needed?
attach(rounded)

conf <- table(actual, prediction)
```

```
TP <- conf[1,1]
TN <- conf[2,2]
FP <- conf[1,2]
FN <- conf[2,1]

Accuracy <- round((TP + TN) / (TP + TN + FP + FN), digits = 3)
Precision <- round((TP / (TP + FP)), digits = 3)
Recall <- round((TP / (TP + FN)), digits = 3)
FScore <- round((2 * Precision * Recall)/(Precision + Recall), digits = 3)

train_Accuracy <- Accuracy
train_Precision <- Precision
train_Recall <- Recall
train_FScore <- FScore
```

On the training dataset, the NN correctly predicted 711 true positives and 103 true negatives. It missed by 14 with the false positives and 24 with the false negatives. The model's accuracy came to 0.955, precision 0.981, recall 0.967 and FScore reached 0.974.

Are you pleased with the result? If your answer was either Yes or No, then you are wrong. The correct response would be Perhaps. It's simply too soon to say, since we haven't yet considered whether this model has been overfit. We need to see how the model performs on the test dataset. Our current parameters (weights) used to model the training dataset may not perform well on out-of-sample observations. To counter overfitting with NNs, we should provide more observations and adjust the model's threshold.

Testing dataset

Having explored the models fit to the training dataset, we now explore its ability to generalize to the test dataset. This is merely repeating the above steps above, but replacing the dataset.

```
# We subset to remove Y from the dataset
temp_testing <- subset(testing, select = c("X1","X2", "X3", "X4", "X5", "X6", "X7", "X8", "X9", "X10", "X11", "X12", "X13"))

fit.results <- compute(fit, temp_testing)

results <- data.frame(actual = testing[,14], prediction = fit.results$net.result)

# Display first 5 rows
results[1:10,]
```

```
##    actual         prediction
## 1       1 0.0020544526684816
## 2       0 0.0000002925049011
## 3       0 0.0000290038395372
## 4       0 0.0000002920340727
## 5       0 0.0000002920221245
## 6       0 0.0000003145796196
## 7       1 0.9943735815568385
## 8       0 0.0000003061777268
## 9       1 0.0001258777899207
## 10      0 0.0000002920221961
```

Accuracy, Precision, Recall, FScore

```
rounded <- sapply(results, round, digits = 0)
rounded = data.frame(rounded) #redefine as df

# Is this needed?
attach(rounded)
```

```
## The following objects are masked from rounded (pos = 3):
##
##     actual, prediction
```

```
conf <- table(actual, prediction)

TP <- conf[1,1]
TN <- conf[2,2]
FP <- conf[1,2]
FN <- conf[2,1]

Accuracy <- round((TP + TN) / (TP + TN + FP + FN), digits = 3)
Precision <- round((TP / (TP + FP)), digits = 3)
Recall <- round((TP / (TP + FN)), digits = 3)
FScore <- round((2 * Precision * Recall)/(Precision + Recall), digits = 3)

test_Accuracy <- Accuracy
test_Precision <- Precision
test_Recall <- Recall
test_FScore <- FScore
```

On the test set, the NN correctly predicted 283 true positives and 52 true negatives. It missed by 8 with the false positives and 21 with the false negatives.

The model's accuracy came to 0.92, precision 0.973, recall 0.931 and FScore reached 0.952.

Training vs Testing

Accuracy
train_Accuracy = 0.92
test_Accuracy = 0.955
Δ = 0.035

Precision
train_Precision = 0.973
test_Precision = 0.981
Δ = 0.008

Recall
train_Recall = 0.931
test_Recall = 0.967
Δ = 0.036

FScore
train_FScore = 0.952
test_FScore = 0.974
Δ = 0.022

ROC Curve

Accuracy increased??????????????

How calculate RMSE???????????

How do I determine Accuracy, etc for both classes??????????

Finally, we'll construct an ROC curve??????????

Results

INCOMPLETE From the results, we see that 1,032 bacterium were *Flagged*, and 184 were *Not Flagged* for further study. This is slightly moved from our original dataset where 1,017 were *Flagged*, and 200 were *Not Flagged* for further study.

Stopwatch

Click the stopwatch off.

```
#stop watch
end_time <- Sys.time()
total_time <- end_time - start_time
total_time = round(total_time, digits = 2)
```

The total time required to process all rows into two classes was 7.73 seconds. Of that total time, 4.08 seconds was used to fit the model.

Thank you for reading, and happy Neural Networking!

Data

If you don't have the dataset, copy the table below, and after pasting it into Excel, save it as a comma separated file named patients.csv in your preferred directory.

X1	X2	X3	X4	X5	X6	X7	X8	X9	
-0.9383188968	-1.8122910585	0.4232758667	-0.9326550200	-0.8166931491	0.0196864816	1.4996027097	-0.9087278786	0.2332976278	-1.220764
0.4100449216	-1.6761346421	0.5542687697	0.2003773930	0.1337986949	0.5017105291	1.6831905586	0.3741902068	0.2332976278	0.598057
-0.5478091570	-1.6194521131	1.8785362114	-1.0581368865	0.9386461351	-6.2016990394	1.2519757821	-0.5371723675	0.2332976278	-0.566766
0.1816335644	-1.5275842269	2.3343939171	0.5780980307	-0.4175693567	0.6819317505	-1.0503096979	0.1568652852	0.2332976278	0.354116
0.2405784308	-1.5248036550	0.4813763655	0.3472180313	-0.0331238735	0.5796732601	1.5834211790	0.2129491359	0.2332976278	0.418599
0.6458243872	-1.4398124789	-0.1673252998	0.4600406900	-0.0531007182	0.6078207727	-0.0170738674	0.5985256097	0.2332976278	0.834817
0.4395173548	-1.4738526162	0.2467806950	0.0877931774	0.9468207619	0.2000652712	1.5374045268	0.4232635762	-0.7316340574	0.628431
0.2258422142	-1.4567861625	2.3009192601	0.3253291582	-0.3650904975	0.6358602422	-0.9568389148	0.2059386546	-0.2491682148	0.402583
-0.7393799728	-1.4805425272	-0.3138166124	-0.7016624152	-0.2162328681	-0.0551609921	-1.7637336340	-0.7194448824	0.2332976278	-0.865587
-0.9309507885	-1.4847817268	2.0797087125	-1.0036634473	-0.6885350899	-0.3096233747	-1.6270720044	-0.9017173973	0.2332976278	-1.206537
-0.7246437562	-1.4698392858	-0.0424889307	-1.0433569798	0.4575751208	-3.0844902293	-1.7584122320	-0.6984134384	-0.2491682148	-0.841314
1.6478871156	-1.4007585181	0.1377994142	1.7022205224	0.9025680754	0.6564937069	1.7564366702	1.5589615534	-0.2491682148	1.718053
-0.0467777928	-1.3155467624	2.5073250037	0.2542072340	-0.6064143891	0.6640395821	-0.7477375512	-0.0604596364	0.2332976278	0.092706
0.0563757234	-1.3157842662	-0.3247995271	-0.0541565370	0.1364665934	0.3938136080	1.3345285427	0.0376871024	0.2332976278	0.213168
-0.0394096845	-1.3309404449	0.0129766063	0.5763496101	-0.8024636611	0.7280687966	1.7481896900	-0.0534491550	0.2332976278	0.101452
0.3658362719	-1.2739717855	-0.0752141835	0.9957272201	0.9533686346	0.5495946219	-1.3338508506	0.4162530948	-1.6965657427	0.552047
0.4468854631	-1.2680601820	0.2850658604	0.8546056229	-0.1252231196	0.6777809456	1.6695216740	0.4092426135	0.2332976278	0.635988
1.1984325095	-1.0521217756	-1.1222304294	1.0956221484	0.4710489093	0.6344134096	-0.6765332319	1.1663745983	-0.2491682148	1.342701
-0.0983545509	-1.1281435372	1.7542891143	0.0944876354	0.0589363326	0.4827558158	-1.6865143118	-0.1095330058	0.2332976278	0.030844
-0.5478091570	-1.0957551913	-1.0581716847	-0.8133286635	-0.0460087278	-0.4756487303	-0.7142974443	-0.5371723675	0.2332976278	-0.566766
0.2405784308	-1.0511082329	-0.0685724753	-0.5619931263	1.1938912019	-1.0228404198	-0.0921596849	0.2269700986	-0.2491682148	0.418599
0.0490076151	-1.0609769718	1.7072831551	0.1155262493	-0.3572237615	0.5887417512	-1.7268420324	0.0306766211	0.2332976278	0.204701
-0.4814961824	-1.0477155523	1.3378271970	-0.5403669609	-0.2998794457	0.2204126242	-1.7503403266	-0.4740780355	0.2332976278	-0.470659
-0.4372875326	-0.9901497178	-1.2557147122	-0.6628374327	0.0383504887	-0.1863579001	1.2398941978	-0.4320151474	0.2332976278	-0.408359
-0.8572697055	-0.9361696929	-0.2125850400	-0.7276007291	-0.9736472432	0.4083197443	0.4540710837	-0.8316125839	0.2332976278	-1.069241
-0.1425632007	-0.9248134190	-0.1278170868	-0.0883299578	-0.3643578230	0.5281297472	0.5741035372	-0.1515958938	0.2332976278	-0.023107
-0.3562383413	-0.9115218459	-0.7644896826	-0.4517771518	0.0460734115	0.1256158812	-1.4313456850	-0.3548998527	0.2332976278	-0.297491
2.7531033602	-0.8803976166	-0.4183799217	2.1046062771	1.5983810435	0.6303984022	-1.6163058422	3.0802360044	-0.7316340574	2.539411
-0.3636064496	-0.8993605172	-1.0983029609	-0.1877300970	-0.5349819622	0.5388792786	-1.5183579747	-0.3619103340	0.2332976278	-0.307400