Milestone 1

Andres Calderon
acald013@ucr.edu

January 25, 2016

1 Introduction

This report describes the initial steps in order to accomplish the final project in the course. The main goal of the project is to perform a reliability analysis of a machine learning algorithm (kNN). During this first milestone it is intended to provide a flexible implementation of the kNN algorithm, a mechanism to inject random errors in the calculation of the distance metric and a framework to measuring the impact of an unreliable distance calculation.

2 A flexible implementation of kNN

There are many open source implementations of the kNN algorithm under different programming languages. This report uses the R Project for Statistical Computing[1] platform using specifically the **knnflex**[2] package. The **knnflex** package allows a more flexible implementation of the distance metric as well the opportunity to code custom functions for aggregations and tie handlers. In addition, it uses the **caret**[3] package to compute the confusion matrix and associated statistics for the model fit.

2.1 A quick classification example

The code in figure 1 illustrates the use of **knnflex** to classify a small random set of features. In lines 5 to 10 it sets the number of instances and a random seed, create two attributes with random numbers (x1 and x2) and a binary class (y). Lines 11 and 12 split the data set in training and testing set (75% and 25% respectively). Line 17 call the `kdd.dist` function which will generate a distance matrix among all the instances in the data set. Line 18 perform the classification calling the `knn.predict` function. It takes the training and testing data sets,

[1] https://www.r-project.org/

[2] `http://ftp.uni-bayreuth.de/math/statlib/R/CRAN/src/contrib/Descriptions/knnflex.html`

[3] `https://cran.r-project.org/web/packages/caret/index.html`

```
1  require(knnflex)
2  require(caret)
3
4  # a quick classification example
5  n <- 200
6  set.seed(123)
7  x1 <- c(rnorm(n/2,mean=2.5),rnorm(n/2,mean=7.5))
8  x2 <- c(rnorm(n/2,mean=7.5),rnorm(n/2,mean=2.5))
9  x  <- cbind(x1,x2)
10 y <- c(rep(1,n/2),rep(0,n/2))
11 train <- sample(1:n,n*0.75)
12 test <- (1:n)[-train]
13 # plot the training cases
14 plot(x1[train],x2[train],col=y[train]+1,xlab="x1",ylab="x2"
15      ,xlim=c(-1,10),ylim=c(-1,10))
16 # predict the other cases
17 kdist <- knn.dist(x)
18 preds <- knn.predict(train,test,y,kdist,k=3,agg.meth="majority")
19 # add the predictions to the plot
20 points(x1[test],x2[test],col=as.integer(preds)+1,pch="+")
21 # display the confusion matrix
22 confusionMatrix(y[test],preds)
```

Figure 1: A quick code example

the distance matrix, the number of neighbors to be taken into account and the aggregation method to pick the class between them.

Finally, line 22 calls the `confusionMatrix` function to retrieve the accuracy and other statistics from the model (figure 2). Lines 14 and 20 plot the initial instances in the training set and the result of the classification for the instances in the testing set. Figures 3 and 4 show the results respectively.

3 Error injector

The code in figure 5 takes the distance matrix generated in line 17 of figure 1 and introduces random errors. A distance value is changed according to a probability passed as second parameter. By default this value is set to 2%.

For example, given the small 4×4 distance matrix shown in figure 6, it can be seen the effect of the `injectError` function. It shows that in position (1,2) an error was injected.

4 Simulations

The code in figure 7 is used to perform a simulation using the iris[4] data set. The original accuracy is stored in the line 13. Then, it runs 1000 iterations where

[4] `https://archive.ics.uci.edu/ml/datasets/Iris`

```
confusionMatrix(y[test],preds)

## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 27  0
##          1  0 23
##
##                Accuracy : 1
##                  95% CI : (0.9289, 1)
##     No Information Rate : 0.54
##     P-Value [Acc > NIR] : 4.166e-14
##
##                   Kappa : 1
##  Mcnemar's Test P-Value : NA
##
##             Sensitivity : 1.00
##             Specificity : 1.00
##          Pos Pred Value : 1.00
##          Neg Pred Value : 1.00
##              Prevalence : 0.54
##          Detection Rate : 0.54
##    Detection Prevalence : 0.54
##       Balanced Accuracy : 1.00
##
##        'Positive' Class : 0
##
```

Figure 2: Confusion matrix

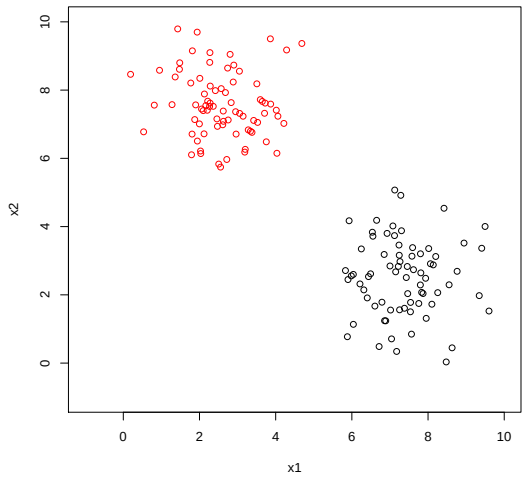

Figure 3: Instances in training set.

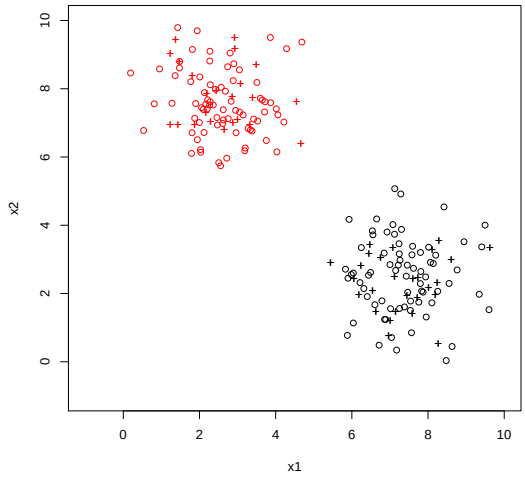

Figure 4: Results for instances in testing set.

```
injectError <- function(kdist, p = 0.02){
  n <- nrow(kdist)
  m <- n
  mn <- .Machine$double.xmin
  mx <- .Machine$double.xmax
  for(i in 1:(m-1)){
    for(j in (i+1):n){
      if(runif(1) < p){
        random <- runif(1, mn, mx)
        kdist[i,j] <- random
        kdist[j,i] <- random
        # print(paste(i,j))
      }
    }
  }
  return(kdist)
}
```

Figure 5: Code for error injection

```
> kdist
         1        2        3        4
1 0.000000 1.619811 8.513312 8.520310
2 1.619811 0.000000 9.230566 9.580171
3 8.513312 9.230566 0.000000 2.278977
4 8.520310 9.580171 2.278977 0.000000

> injectError(kdist)
[1] "1 2"
              1             2        3        4
1  0.000000e+00 8.544733e+307 8.513312 8.520310
2 8.544733e+307  0.000000e+00 9.230566 9.580171
3  8.513312e+00  9.230566e+00 0.000000 2.278977
4  8.520310e+00  9.580171e+00 2.278977 0.000000

>
```

Figure 6: Error injection example

errors are injected to the distance matrix and new predictions are performed. The accuracies for the new runs are stored in an array (line 22).

The last 5 lines display a summary of statistics and plot the results (figures 8 and 9). From figure 8 we can see that the median of the accuracies during the run is the same that the original accuracy, the standard deviation of the run was just 0.003 and in 92.6% of the cases kNN still gives the correct results. Figure 10 shows the density distribution of the run.

```r
require(knnflex)
require(caret)

n <- nrow(iris)
x <- iris[,1:4]
y <- iris[,5]
train <- sample(1:n,n*0.5)
test <- (1:n)[-train]
kdist <- knn.dist(x)
preds <- knn.predict(train,test,y,kdist)
cm <- confusionMatrix(y[test],preds)

accuracy_orig <- cm$overall['Accuracy']

runs <- 1000
accuracy_error <- c()
for(i in 1:runs){
  kdist_error <- injectError(kdist, 0.02)
  preds_error <- knn.predict(train,test,y,kdist_error)
  cm_error <- confusionMatrix(y[test],preds_error)

  accuracy_error <- c(accuracy_error, cm_error$overall['Accuracy'])
}

print(paste("Original accuracy=",accuracy_orig))
print(summary(accuracy_error))
print(sd(accuracy_error))
print(length(accuracy_error[accuracy_error == accuracy_orig]) / runs)
plot(accuracy_error, cex=0.5,xlab="Run",ylab="Accuracy")
abline(h = accuracy_orig, col='red', cex=0.2)
```

Figure 7: Simulation in the iris data set

```
> source('simulation.R')
[1] "Original accuracy= 0.933333333333333"
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.9200  0.9333  0.9333  0.9343  0.9333  0.9600
[1] 0.003571311
[1] 0.926
>
```

Figure 8: Error injection example

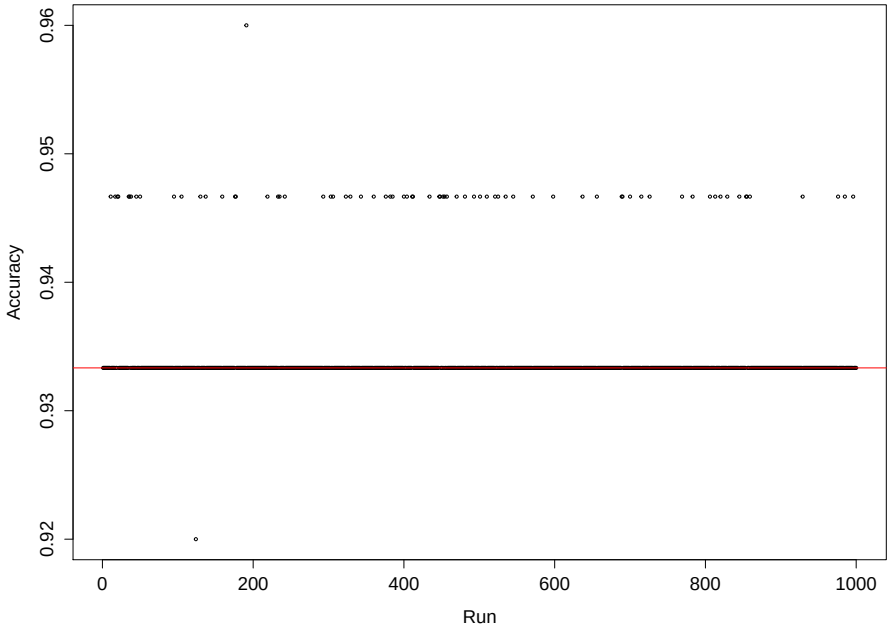

Figure 9: Accuracies after error injection. Red line shows the original accuracy.

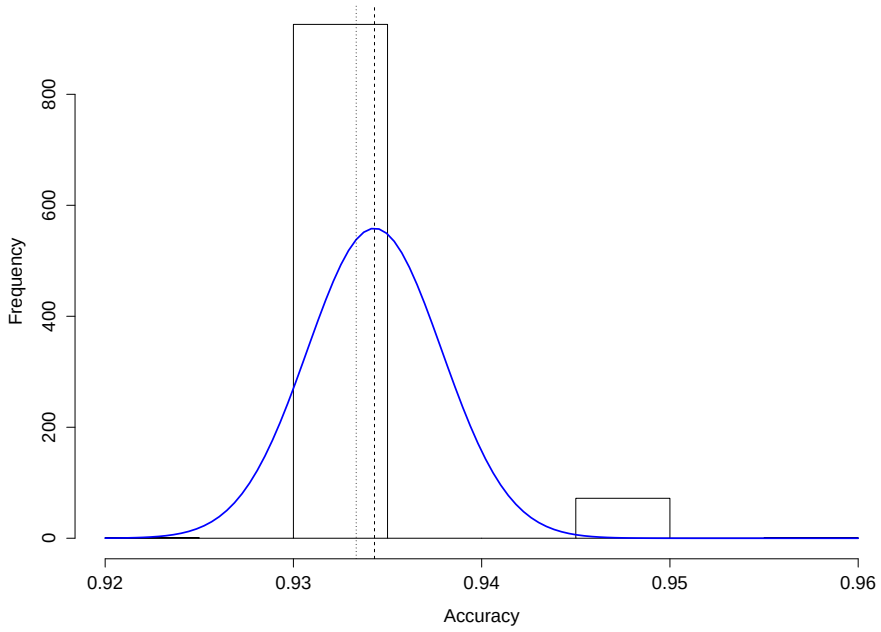

Figure 10: Density distribution of the run. Dotted line is original accuracy, dashed line is mean accuracy in the run.