MATH 6350 Fall 2020 MSDS
Homework 2

Sara Nafaryeh
Tony Nguyen
Thomas Su

All authors played equal part

Introduction

In this report we will select three font data sets consisting of digitized images of typed characters and determine using the KNN function in R to predict our best K.

The files were downloaded from a zip file from the following link: *https://archive.ics.uci.edu/ml/machine-learning-databases/00417/*

The font types that we chose where: COMIC, BOOKMAN, and MONOTYPE. Each has 412 column features along with total observed cases of 2669, 2388, and 2388 respectively. Each case describes numerically a digitized image of some specific character typed in each of the fonts. The images have 20x20 = 400 pixel sizes, each with its own "gray level" indicated by an integer value of 0 to 255. All the fonts have 412 feature columns, where 400 of them describe the 400 pixels named:

{ r0c0, r0c1,r0c2, … , r19,c17, r19c18, r19c19}

"rLcM" = gray level image intensity for pixel in position **{Row L, Column M}**.

Finally, the last important columns in the data set we will be looking at are the strength and italic columns. The strength column lists 0.4 = normal character and 0.7 = bold character. The italic column lists 1 = italic character and 0 = normal character.

Data Setup

The following steps are taken to get R ready as well as organize our data for the steps that follow. Other than the 400 columns that are associated with the pixels, the data set font files each have the following 12 names:

{ font, fontVariant, m_label, strength, italic, orientation, m_top, m_left, originalH, originalW, h, w }

Of these 12 we need to discard the following 9:

{fontVariant, m_label, orientation, m_top, m_left, originalH, originalW, h, w}

And keep the following 3: ***{font, strength, italic}*** as well as the 400 pixel columns named: ***{ r0c0, r0c1,r0c2, … , r19,c17, r19c18, r19c19}*** therefore we are left with 403 columns. After these steps are completed, we define three CLASSES on images of the "normal" character were we extract all the rows in which our three fonts have both strength of 0.4 and italic of 0:

CL1 = all rows of **comic.csv** file for which {strength = 0.4 and italic=0}

CL2 = all rows of **bookman.csv** file for which {strength = 0.4 and italic=0}

CL3 = all rows of **monotype.csv** file for which {strength = 0.4 and italic=0}

And left with the following row outputs:

```
> n1 = nrow(CL1)
> n2 = nrow(CL2)
> n3 = nrow(CL3)
> N = sum(n1, n2, n3)
> n1;n2;n3;N
[1] 597
[1] 667
[1] 667
[1] 1931
```

The respected row sizes for CLASS1, CLASS, and CLASS3 are named

n1, n2, n3 = 597, 667, 667 and there sum → N = 1931 cases

We combine them all together into a data set named DATA using the function **rbind()** and see it has dimensions of 1931 rows and 403 columns: the 403 being font, strength, italic, +400 pixels we will call features X1, X2, … X400. Each such feature Xj is observed N times, and its N observed values are listed in the column "j" of DATA.

```
> DATA = rbind(CL1, CL2, CL3)
> str(DATA)
'data.frame':	1931 obs. of  403 variables:
 $ font    : chr  "COMIC" "COMIC" "COMIC" "COMIC" ...
 $ strength: num  0.4 0.4 0.4 0.4 0.4 0.4 0.4 0.4 0.4 0.4 ...
 $ italic  : num  0 0 0 0 0 0 0 0 0 0 ...
 $ r0c0    : num  1 1 1 1 1 1 255 255 255 1 ...
 $ r0c1    : num  1 1 1 1 1 1 255 255 213 1 ...
 $ r0c2    : num  1 1 1 1 1 1 255 255 213 1 ...
 $ r0c3    : num  1 1 1 1 1 1 255 255 213 1 ...
 $ r0c4    : num  1 1 1 1 1 1 255 255 213 1 ...
 $ r0c5    : num  1 1 1 1 1 1 255 255 213 1 ...
 $ r0c6    : num  1 1 1 175 86 1 255 255 213 1 ...
 $ r0c7    : num  1 17 1 255 255 33 255 255 213 1 ...
 $ r0c8    : num  48 86 1 255 255 215 255 255 213 1 ...
 $ r0c9    : num  83 166 83 255 255 255 255 255 213 1 ...
 $ r0c10   : num  169 206 255 255 255 154 255 255 213 1 ...
 $ r0c11   : num  169 255 255 255 255 9 255 255 213 1 ...
 $ r0c12   : num  201 255 255 255 255 1 255 255 213 1 ...
 $ r0c13   : num  255 255 255 175 86 1 255 255 213 1 ...
 $ r0c14   : num  255 255 255 1 1 1 255 255 213 1 ...
 $ r0c15   : num  255 126 198 1 1 1 255 255 213 73 ...
 $ r0c16   : num  255 9 120 1 1 1 255 255 213 255 ...
 $ r0c17   : num  255 1 14 1 1 1 255 255 213 255 ...
 $ r0c18   : num  251 1 1 1 1 1 255 255 213 255 ...
 $ r0c19   : num  169 1 1 1 1 1 255 255 255 255 ...
 $ r1c0    : num  1 1 1 1 1 1 255 255 213 1 ...
 $ r1c1    : num  1 1 1 1 1 1 54 255 1 1 ...
 $ r1c2    : num  1 1 1 1 1 1 29 255 1 1 …
 $ r1c3    : num  1 1 1 1 1 1 29 255 1 1 ...
 $ r1c4    : num  1 1 1 191 126 1 29 255 1 1 ...
 $ r1c5    : num  1 1 1 242 227 1 29 255 1 1 ...
 $ r1c6    : num  67 86 1 251 237 19 29 255 1 1 ...
 $ r1c7    : num  174 225 1 103 255 185 29 255 1 1 ...
 $ r1c8    : num  254 255 1 14 255 255 29 255 1 1 ...
  [list output truncated]
```

Part 0

The ***start_col*** and ***end_col*** are defined to be the feature X1 starts and X400 ends respectfully. Now that we have created our [1931x403] ***DATA*** set table, we observe the mean, m = mean(X1) ....mean(X400), and standard deviation, s = sd(X1) ....sd(400) for each of our pixel features 1 to 400. The following output shows the first 6 values of the mean and standard deviation using head() function in R:

```
> head(m)
 r0c0         r0c1         r0c2         r0c3         r0c4         r0c5
42.24392      58.08597     63.99845     64.72139     69.71362     77.50492
> head(s)
r0c0          r0c1         r0c2         r0c3         r0c4         r0c5
85.60722      100.04134    104.25599    101.38599    102.82648    105.25836
```

The most important step to do before running our KNN() function is to standardize the data features in order to set everything at a mean of 0 and standard deviation at 1. It is important to standardize because when we compare measurements they can have different units as one can observe in the table above. In the table below observe the variance of the data before and after standardization. Notice the variance of the DATA of X1 and X2 features before, and after when they all equal to 1. Variables that are measured at different scales do not contribute equally to the analysis and might end up creating a bais. We scale the data and name it SDATA:

```
> var(DATA[,4]); var(DATA[,5])
[1] 7328.596
[1] 10008.27
> SDATA = scale(X)
> var(SDATA[,4]); var(SDATA[,5])
[1] 1
[1] 1
```

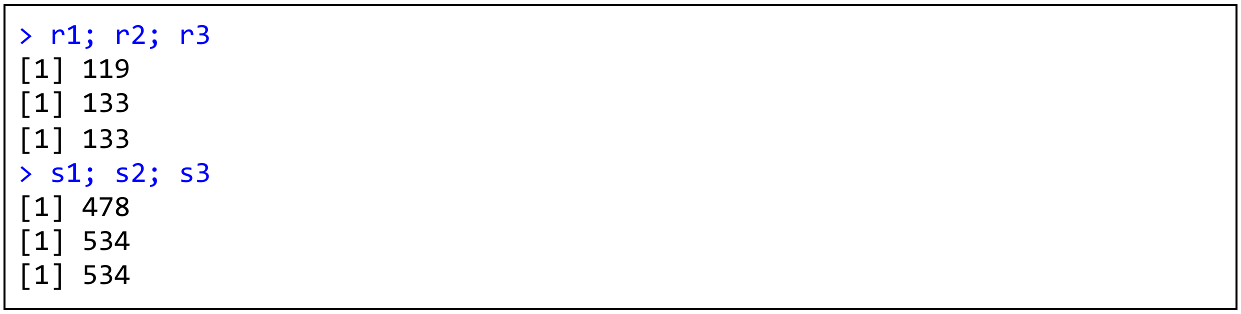
Finally, we compute the 400x400 correlation matrix CORR of the 400 features X1, X2,... ...X400 Extract the 10 pairs Xi,Xj of features which have the 20 highest absolute values |CORR(i,j)| and display these 20 top highest correlation values with is corresponding pixel positions pair Xi and Xj

```
# There are the top 20 highest absolute correlation values except 1 as diag(r), and each value has two
amounts. The output below doesn't repeat the same numbers.
[1] 0.9238824
[1] 0.9227696
[1] 0.9196329
[1] 0.9188532
[1] 0.917081
[1] 0.9167512
[1] 0.915772
[1] 0.9153191
[1] 0.9140681
[1] 0.9111543

# Because the correlation matrix is symmetric, the locations of the top 20 highest absolute values in the
correlation matrix are (i, j) and (j, i). For example, (3, 2), (2, 3), (391, 390), (390, 391)......
[1] 3 2
[1] 391 390
[1] 244 224
[1] 259 239
[1] 399 398
[1] 243 223
[1] 262 242
[1] 279 259
[1] 239 219
[1] 223 203
```

PART 1.0

For our KNN() function we need to split the data into 80% train set and 20% test set. Notice, before we do the split we have to set.seed() in order to ensure the same starting values are used throughout the experiment if we start with the same seed, set.seed(1). Splitting the sets are done individually for each CLASSES we defined earlier in the data setup step. Among the ***n1*** = 597 rows of class CL1, we select a number **r1** = 20% of ***n1*** and name this as ***testCL1***. The remaining rows are named **s1** = ***n1-r1*** to be our 80% train set for class CL1 defined as ***trainCL1***. The approach is done for the test and train sets of classes CL2 and CL3 respectfully. The following output represents the number of row cases for each class CL1, CL2, CL3 respectfully :



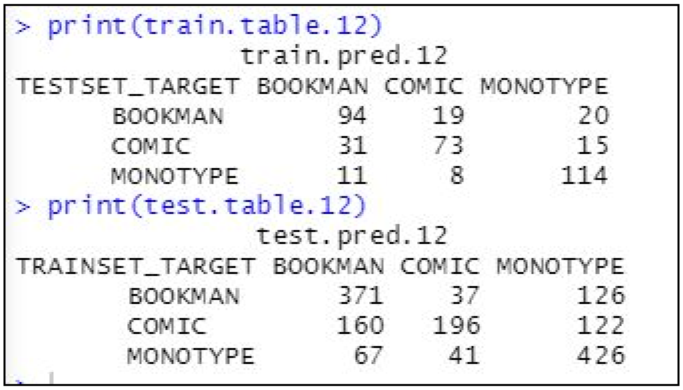
```
> r1; r2; r3
[1] 119
[1] 133
[1] 133
> s1; s2; s3
[1] 478
[1] 534
[1] 534
```

After we have spit the data into test and train sets we can define the full training and test as our prediction sets: ***TRAINSET,*** as the union of {*trainCL1, trainCL2, trainCL3*}. A matrix with 1546 rows x 400 columns and the full test set: ***TESTSET***, as the union of {*testCL1, testCL2, testCL3*}. A matrix with 385 rows x 400 columns. The true train defined as ***TRAINSET_TARGET***, is a string vector [1546x1] and The true defined as ***TESTSET_TARGET***, is a string vector [385x1] that indicates the true class values for each of out classes COMIC, BOKMAN, and MONOTYPE. We use the predictions to observe whether or not they hold TRUE or FALSE based upon the test and training true target values.

Part 1.1

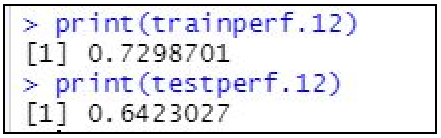
Before we can run the program for a fix K = 12 we need to make sure we set.seed(1) as well as call the library(class) since KNN is part of R library package labeled class. Using the standardized data matrix SDATA, from Part 0, to apply the K nearest neighbor KNN algorithm. We define our predicted training knn at k =12 as: ***train.pred.12*** where the train = ***TRAINSET***, test = ***TESTSET***, and the true train = cl = ***TRAINSET_TARGET.***

We define our predicted testing knn at k =12 as **test.pred.12** where the train = ***TESTSET***, test = ***TRAINSET***, and the true train = cl = ***TESTSET_TARGET.*** The following output shows ***train.table.12*** where we compare the ***train.pred.12*** vs ***testset_target***, and ***test.table.12*** where we compare the ***test.pred.12*** *vs* ***TRAINSET_TARGET***



```
> print(train.table.12)
             train.pred.12
TESTSET_TARGET BOOKMAN COMIC MONOTYPE
      BOOKMAN       94    19       20
      COMIC         31    73       15
      MONOTYPE      11     8      114
> print(test.table.12)
              test.pred.12
TRAINSET_TARGET BOOKMAN COMIC MONOTYPE
       BOOKMAN      371    37      126
       COMIC        160   196      122
       MONOTYPE      67    41      426
```

Overall, looking at both our tables for train and test, we notice high values along the diagonal and relatively low everywhere else, which we hope to see. This shows that the percentage of the sets we choose to predict are among the true values of the classes at fixed k =12. Below, defined as ***trainperf.12*** and ***testperf.12*** we compute the percentages of correct classification from the tables above.



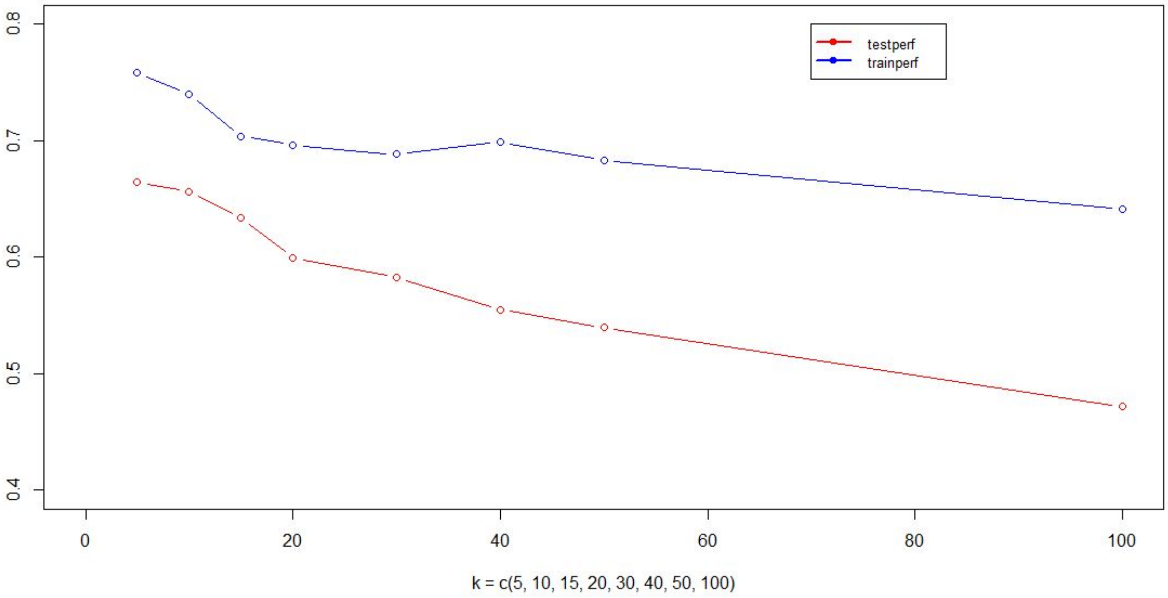
```
> print(trainperf.12)
[1] 0.7298701
> print(testperf.12)
[1] 0.6423027
```

The percentages of correct classifications of ***trainperf.12*** = 72.99% on the ***TRAINSET***. This shows that the number of font classes we predicted for that class to be the true value came out to be 72.99% correct for the trainset.

The percentage of correct classification of ***testperf.12*** = 64.23% on the ***TESTSET***. This shows that the number of font classes we predicted for the three classes to be true was 64.23% for the testset. These values are moderate, but not strong. We will have to test some more k values to find the best k nearest neighbor for our data set.

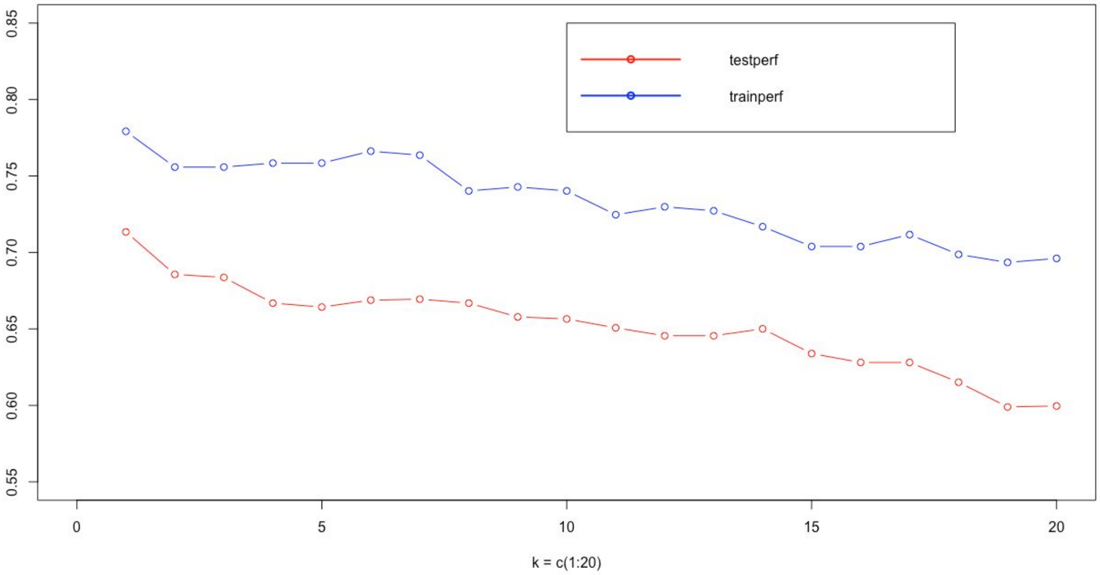
Part 1.2 & 1.3

In this step we will repeat the preceding operation for k = 5, 10, 15, 20, 30, 40, 50,100; using the define function ***testperf*** to compute the associated percentages ***testperf*** of correct classifications on the ***TESTSET***. The graph below is a plot of both accuracies of the ***testpref*** in red and the ***trainperf*** in blue. This is done to observe, if any, overfitting and help us determine the "best k" to be selected.



After plotting the curve ***testperf_k*** versus ***k***, it seems likely the best range of values for the integer K is [1 ≤ K ≤ 20].

The best value k ***"kbest"*** for the integer K should be selected by the intersect of ***testperf_k*** and ***trainperf_k***, because we don't expect either overfitting or underfitting.



We repeat the preceding exploration for all the values of K in the range [1, 20] and graph them together for a better visualization. Based on the graph above, we can conclude the training set has a better performance than the test set does at all times, which means there is overfitting without the optimal point as known as the ideal ***"kbest"***. But, in this situation, we picked the ***"kbest"*** to be 1, where two curves have the minimum discrepancy, and also have the best accuracy of both.

Part 1.4

Using the ***"kbest"*** = 1, we will compute and interpret the two 3x3 confusion matrices for KNN classification, namely the matrices ***testconf.kbest*** on the ***TESTSET*** and ***trainconf.kbest*** on the ***TRAINSET***: First, we initialize set.seed(1). Starting with the ***TESTSET*** and using the built-in knn function in R with our parameters in our case as: train = ***TESTSET***, test = ***TRAINSET***, cl = ***TESTSET_TARGET***, k = ***kbest***, we obtain the ***test.pred.kbest*** and ***test.table.kbest***. Thus, we can calculate the ***testperf.kbest*** = sum(diag(test.table.kbest)) / sum(test.table.kbest). After all, we get the confusion matrix for the ***testconf.kbest***:

```
> print(testconf.kbest) # confusion matrix for test set
                   test.pred.kbest
TRAINSET_TARGET    BOOKMAN      COMIC        MONOTYPE
       BOOKMAN     0.76779026   0.12546816   0.10674157
       COMIC       0.27824268   0.60041841   0.12133891
       MONOTYPE    0.16292135   0.07677903   0.76029963

> print(testperf.kbest) # global performance for test set
[1] 0.7134541
```

Following the same procedure, we apply for the ***TRAINSET*** with the ***train.pred.kbest*** = knn(train = ***TRAINSET***, test = ***TESTSET***, cl = ***TRAINSET_TARGET***, k = ***kbest***) and the ***trainperf.kbest*** = sum(diag(train.table.kbest)) / sum(train.table.kbest). Therefore, we obtain the confusion matrix for the ***trainconf.kbest:***

```
> print(trainconf.kbest) # confusion matrix for training set
                  train.pred.kbest
TESTSET_TARGET    BOOKMAN      COMIC        MONOTYPE
      BOOKMAN     0.78195489   0.12781955   0.09022556
      COMIC       0.15966387   0.73109244   0.10924370
      MONOTYPE    0.12781955   0.05263158   0.81954887
```

```
> print(trainperf.kbest) # global performance for training set
[1] 0.7792208
```

The matrix appears to be strong as the diagonal values are high and everything else is low, therefore we can are confident for the ***“kbest”*** =1

Part 1.5

In this part, we compute and interpret confidence intervals for the 3 diagonal terms of the matrix ***testconf.kbest***:

To obtain the confidence intervals, we calculate using the formula following by the function:

CI = p + c(-1, 1) * qnorm((1 + CL)/2) * sqrt(p * (1 - p)/n)

We assume the confidence level is 90% in this case to obtain the confidence intervals for each of ***BOOKMAN, COMIC and MONOTYPE***, respectively:

BOOKMAN:

```
> CI(diag(testconf.kbest)[1], 0.90, nrow(testCL1))
[1] 0.7041231    0.8314574
```

COMIC:

```
> CI(diag(testconf.kbest)[2], 0.90, nrow(testCL2))
[1] 0.5305580    0.6702788
```

MONOTYPE:

```
> CI(diag(testconf.kbest)[3], 0.90, nrow(testCL3))
[1] 0.6994121    0.8211871
```

We can be 90% confident that for every additional unit increase, the testconf fluctuates between [.7041, 0.8314] for BOOKMAN, [0.5305, 0.6702] for COMIC, and [0.6994, 0.8211] for MONOTYPE.

Part 1.6

In this section we are going to observe the percent accuracy of the TEST set by looking at the first 100 features insead of all 400 features we calculated in Part 0. Defined as *PACK1* we considered the 100 attributes with names **rLcM** where **L** = {0, 1, 2,... ,9} and **M** = {0, 1, 2,... ,9}. These 100 features correspond to the 100 pixel intensities displayed in a specific 10x10 window of our 20x20 pixels images, essentially we are taking ¼ th of our total pixels; vectors of 100 features { Z1, Z2, Z3, ... ,Z100 }. Before we can use the KNN function with our "best k" =1 we have to standardize the data defined as *SPACK1*. After that, we need to make sure the values are listed in the 100 specific columns train and test set. To do this we go back to the Data Setup Part from the beginning of the report where we defined **n1** = 597 rows of class CL1, **n2** = 667 rows of CL2, and **n3** = 667 rows of CL3. $\mathbf{r_n}$ = 20% of n = 1,2,3 test splitting and the remaining rows $\mathbf{s_n}$ = 80% of n =1,2,3 train splitting. With these values we set our *TRAINSET.PACK1* and *TESTSET.PACK1*. Finally we can find the percent of correct classification for our test at ***"kbest"*** = 1 defined as: ***w1 = test.pred.kbest1*** . NOTE, that the *TESTSET_TARGECT* we used is from part 1.0

```
> print(test.table.kbest1)
                  test.pred.kbest1
TRAINSET_TARGET   BOOKMAN      COMIC        MONOTYPE
       BOOKMAN    311          75           148
       COMIC      113          272          93
       MONOTYPE   122          56           356
> print(w1)
[1] 0.6073739
```

The percentage of correct classifications ***w1 = testperfKbest*** is observed to be 60.7% achieved by this features package PACK1 on the TESTSET.

Part 1.7

We repeat this operation for 3 other packages PACK2, PACK3, PACK4 of 100 features each corresponding respectively to:

PACK2 of 100 attributes with names rLcM where L = 0, 1, 2,... ,9 and M= 10, 11, 12, ...,19

PACK3 of 100 attributes with names rLcM where L =10,11, 12,... ,19 and M= 0,11,12,...,19

PACK3 of 100 attributes with names rLcM where L =10, 11, 12,...,19 and M= 0, 1, 2,... ,9

This provides three more percentages of correct classifications on the TESTSET:

w2 = testperfKbest achieved by features package PACK2

w3 = testperfKbest achieved by features package PACK3

w4 = testperfKbest achieved by features package PACK4

```
> w2; w3; w4
[1] 0.587969
[1] 0.6701164
[1] 0.6623545
> v = cbind(w1, w2, w3, w4)
> sum(v)/4
[1] 0.6319534
```

The percentage of correct classifications of ***testperfKbest*** at ***w2, w3, w4*** are 58.79%, 67.01%, and 66.23% respectfully. Since each pack is ¼ th of the total 400 pixels, the average equals 63.19% at “kbest” =1.

Part 1.8

In order to assign weights to each feature, namely the weight ***W1*** to each feature in ***PACK1***, the weight ***W2*** to each feature in ***PACK2***, the weight ***W3*** to each feature in ***PACK3***, the weight ***W4*** to each feature in ***PACK4***, we first define a variable as normalizer = 1/sum(w1,w2,w3,w4). By doing so, we can find weight[i] by using the formula:

weight[i] = x[i] * normalizer

Obtaining results:

```
> weight(w1)
[1] 0.2402764
> weight(w2)
[1] 0.2325998
> weight(w3)
[1] 0.2650972
> weight(w4)
[1] 0.2620266
```

Next step, we can assign renormalized weights to each feature by using the formula:

W1 = PACK1/sum(PACK1)*weight(w1)

W2 = PACK2/sum(PACK2)*weight(w2)

W3 = PACK3/sum(PACK3)*weight(w3)

W4 = PACK4/sum(PACK4)*weight(w4)

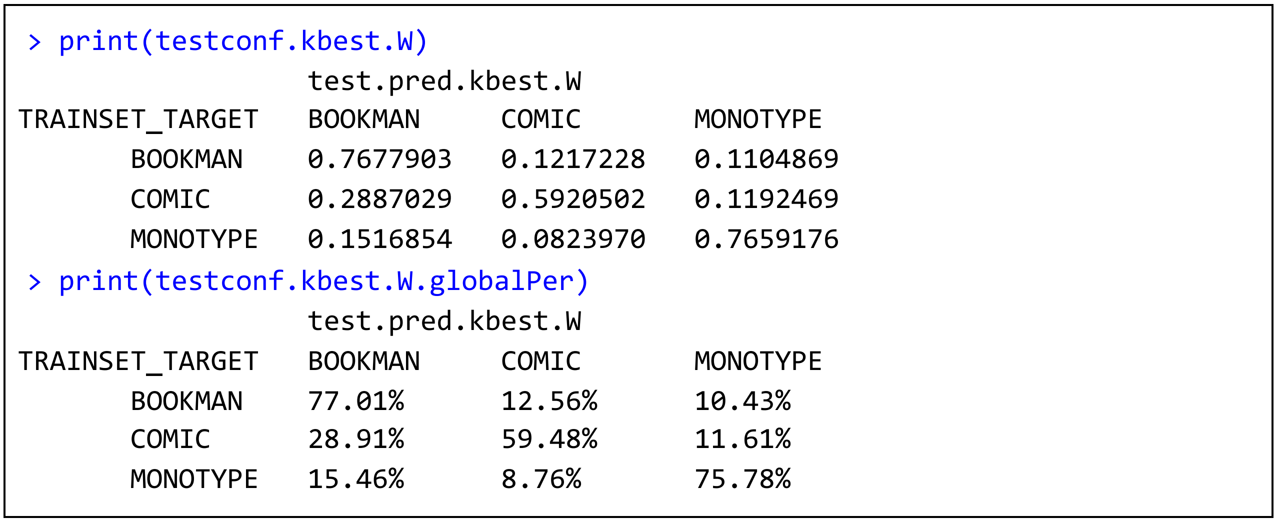
To verify their sum is 1 after renormalizing these 400 weights, we check:

```
> sum(W1,W2,W3,W4)
[1] 1
```

The sum is 1, so we can safely use these 400 weights. Now, we apply KNN with K = ***kbest*** using all 400 features but with the weighted distance defined by these 400 weights. Following the same procedure as Part 1.4, however, we use ***TESTSET.W*** instead of ***TESTSET***. Therefore we have:

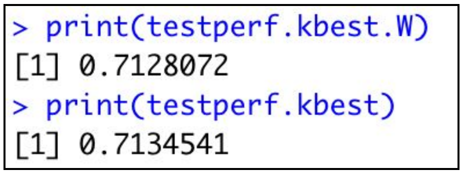
test.pred.kbest.W = knn(train = ***TESTSET.W***, test = ***TRAINSET.W***, cl = ***TESTSET_TARGET***, k = ***kbest***)

Thus, we can obtain the ***test.table.kbest.W*** = table(***TRAINSET_TARGET***, ***test.pred.kbest.W***). We compute the confusion matrix ***test.conf.kbest.W*** for ***TESTSET.W:***



```
> print(testconf.kbest.W)
                    test.pred.kbest.W
TRAINSET_TARGET     BOOKMAN     COMIC       MONOTYPE
        BOOKMAN     0.7677903   0.1217228   0.1104869
        COMIC       0.2887029   0.5920502   0.1192469
        MONOTYPE    0.1516854   0.0823970   0.7659176
> print(testconf.kbest.W.globalPer)
                    test.pred.kbest.W
TRAINSET_TARGET     BOOKMAN     COMIC       MONOTYPE
        BOOKMAN     77.01%      12.56%      10.43%
        COMIC       28.91%      59.48%      11.61%
        MONOTYPE    15.46%      8.76%       75.78%
```

To compare it to the results achieved with the ordinary distance version of KNN, we look at the two results:



```
> print(testperf.kbest.W)
[1] 0.7128072
> print(testperf.kbest)
[1] 0.7134541
```

As our observation, we can see with the weighted distance defined by those 400 weights (***testperf.kbest.W***), the performance increases roughly 0.091% (which is from 71.28% to 71.35%) if compared to the performance of the ordinary distance of KNN (***testpeft.kbest***), which is not improved much because we could not expect any dramatic improvement when the 4 windows are very large and chosen totally arbitrarily. As our suggestion if we want to obtain a significant improvement, we should use many overlapping windows of smaller size and apply a similar approach.

R Codes

```
rm(list=ls()) # to remove all objects from a specified environment
cat("\f") # to clean console

library(readr)
# setwd("C:\\Users\\57299\\Downloads\\fonts") # locate files in Window system
# setwd("~/Library/Mobile Documents/com~apple~CloudDocs/Study Files (Graduate)/MATH
6350/Homework/HW02/fonts/") # locate files in Mac OS
# We selected COMIC.csv, BOOKMAN.csv, MONOTYPE.csv.
COMIC <- data.frame(read_csv("COMIC.csv"))
BOOKMAN <- data.frame(read_csv("BOOKMAN.csv"))
MONOTYPE <- data.frame(read_csv("MONOTYPE.csv"))

c.names = names(COMIC) # All three font files shares same categories of columns

c.discard = match(c("fontVariant", "m_label", "orientation", "m_top", "m_left", "originalH",
              "originalW", "h", "w"),c.names) # discard 9 columns listed

# na.omit() function is to omit all rows that contain NA values
comic = na.omit(COMIC[,-c.discard])
bookman = na.omit(BOOKMAN[,-c.discard])
monotype = na.omit(MONOTYPE[,-c.discard])

CL1 = comic[comic[,match("strength", names(comic))] == 0.4 &
              comic[,match("italic", names(comic))] == 0,]
CL2 = bookman[bookman[,match("strength", names(bookman))] == 0.4 &
               bookman[,match("italic", names(bookman))] == 0,]
CL3 = monotype[bookman[,match("strength", names(monotype))] == 0.4 &
                bookman[,match("italic", names(monotype))] == 0,]
DATA = rbind(CL1, CL2, CL3)
str(DATA)
n1 = nrow(CL1)
n2 = nrow(CL2)
n3 = nrow(CL3)
N = sum(n1, n2, n3)

start_col = match("r0c0",names(DATA))
end_col = ncol(DATA)

X = DATA[,start_col: end_col]
m = sapply(X, mean) # same as apply(X, MARGIN = 2, mean)
s = sapply(X, sd)
SDATA = scale(X)
r = cor(X)

row.location = order(abs(r), decreasing = TRUE)[(length(diag(r)) + 1):(length(diag(r)) + 20)]%/%nrow(r) + 1
column.location = order(abs(r), decreasing = TRUE)[(length(diag(r)) + 1):(length(diag(r)) + 20)]%%nrow(r)
column.location[column.location == 0] = nrow(r)
```

```
max = 0
for (i in 1:10) {
  max = abs(r)[row.location[2*i], column.location[2*i]]
  print(max)
} # There are the top 20 highest absolute correlation values except 1 as diag(r), and each value has two amounts.
The output won’t repeat the same numbers.

max_location = c()
for (i in 1:10) {
  max_location = c(row.location[2*i], column.location[2*i])
  print(max_location)
} # Because the correlation matrix is symmetric, the locations of the top 20 highest absolute values in the correlation
matrix are (i, j) and (j, i). For example, (3, 2), (2, 3), (391, 390), (390, 391)......
```

# part 1.0

```
set.seed(1)
r1 = round(n1 * 0.2) # a set testCL1 of r1 test cases
s1 = n1 - r1 # a set trainCL1 of s1 training cases
r2 = round(n2 * 0.2)
s2 = n2 - r2
r3 = round(n3 * 0.2)
s3 = n3 - r3
set.seed(1)
# a remind that start_col = 4 and end_col = 403
trainCL1 = CL1[1:s1,c(1, start_col:end_col)]
trainCL2 = CL2[1:s2,c(1, start_col:end_col)]
trainCL3 = CL3[1:s3,c(1, start_col:end_col)]
testCL1 = CL1[(s1+1):n1,c(1, start_col:end_col)]
testCL2 = CL2[(s2+1):n2,c(1, start_col:end_col)]
testCL3 = CL3[(s3+1):n3,c(1, start_col:end_col)]
TRAINSET_TARGET = rbind(trainCL1, trainCL2, trainCL3)[,1] # true train set
TESTSET_TARGET = rbind(testCL1, testCL2, testCL3)[,1] # true test set

TRAINSET = rbind(SDATA[1:s1,], SDATA[(n1+1):(n1+s2),], SDATA[(n1+n2+1):(n1+n2+s3),])
TESTSET = rbind(SDATA[(s1+1):n1,], SDATA[(n1+s2+1):(n1+n2),], SDATA[(n1+n2+s3+1):(n1+n2+n3),])
```

# part 1.1

```
library(class)
set.seed(1)
train.pred.12 = knn(train = TRAINSET, test = TESTSET, cl = TRAINSET_TARGET, k = 12) # predict training
test.pred.12 = knn(train = TESTSET, test = TRAINSET, cl = TESTSET_TARGET, k = 12) # predict test
train.table.12 = table(TESTSET_TARGET, train.pred.12)
test.table.12 = table(TRAINSET_TARGET, test.pred.12)
# trainconf.12 = prop.table(train.table.12, margin  = 1) # confusion matrix for training set
# testconf.12 = prop.table(test.table.12, margin  = 1) # confusion matrix for test set
trainperf.12 = sum(diag(train.table.12))/sum(train.table.12) # percentage of correct classifications on TRAINSET
testperf.12 = sum(diag(test.table.12))/sum(test.table.12) # percentage of correct classifications on TESTSET
print(train.table.12)
print(trainperf.12)
print(test.table.12)
print(testperf.12)

# part 1.2
testperf = function(k) {
  set.seed(1)
  test.pred = knn(train = TESTSET, test = TRAINSET, cl = TESTSET_TARGET, k = k)
  test.table = table(TRAINSET_TARGET, test.pred)
  testperf = sum(diag(test.table))/sum(test.table)
  return(testperf)
}
trainperf = function(k) {
  set.seed(1)
  train.pred = knn(train = TRAINSET, test = TESTSET, cl = TRAINSET_TARGET, k = k)
  train.table = table(TESTSET_TARGET, train.pred)
  trainperf = sum(diag(train.table))/sum(train.table)
  return(trainperf)
}
k = c(5, 10, 15, 20, 30, 40, 50, 100)
testperf_k = mapply(testperf, k)
trainperf_k = mapply(trainperf, k)
plot(k, testperf_k, type = "b", col = "red", xlab = "k = c(5, 10, 15, 20, 30, 40, 50, 100)",
     ylab = "", main = "", xlim = c(0, 100), ylim = c(0.4, 0.8))
lines(k, trainperf_k, type = "b", col = "blue")
legend(70, 0.8, c("testperf", "trainperf"), lwd = 2,
       col = c("red", "blue"), pch = 1, cex = 0.8)

k2 = seq(1, 20, 1)
testperf_k2 = mapply(testperf, k2)
trainperf_k2 = mapply(trainperf, k2)
plot(k2, testperf_k2, type = "b", xlab = "k = c(1:20)",
     ylab = "", main = "", col = "red",xlim = c(0, 20), ylim = c(0.55, 0.85))
lines(k2, trainperf_k2, type = "b", col = "blue")
legend(10, 0.85, c("testperf", "trainperf"), lwd = 2,
       col = c("red", "blue"), pch = 1)

# part 1.3
kbest = 1 # where two curves have the minimum discrepancy
print(kbest)

# part 1.4
set.seed(1)
train.pred.kbest = knn(train = TRAINSET, test = TESTSET, cl = TRAINSET_TARGET, k = kbest)
test.pred.kbest = knn(train = TESTSET, test = TRAINSET, cl = TESTSET_TARGET, k = kbest)
train.table.kbest = table(TESTSET_TARGET, train.pred.kbest)
test.table.kbest = table(TRAINSET_TARGET, test.pred.kbest)
trainperf.kbest = sum(diag(train.table.kbest))/sum(train.table.kbest)
```

```
testperf.kbest = sum(diag(test.table.kbest))/sum(test.table.kbest)

trainconf.kbest = prop.table(train.table.kbest, margin  = 1)
print(trainconf.kbest) # confusion matrix for training set
testconf.kbest = prop.table(test.table.kbest, margin  = 1)
print(testconf.kbest) # confusion matrix for test set

# part 1.5
diag(testconf.kbest)
CI = function(p, CL, n) {
  p + c(-1, 1) * qnorm((1 + CL)/2) * sqrt(p * (1 - p)/n)
}
# assume the confidence level is 90%
CI(diag(testconf.kbest)[1], 0.90, nrow(testCL1))
CI(diag(testconf.kbest)[2], 0.90, nrow(testCL2))
CI(diag(testconf.kbest)[3], 0.90, nrow(testCL3))

# part 1.6
# a reminder that DATA = rbind(CL1, CL2, CL3)
#                 TRAINSET_TARGET = rbind(trainCL1, trainCL2, trainCL3)[,1] # true train set
#                 TESTSET_TARGET = rbind(testCL1, testCL2, testCL3)[,1] # true test set
a = function(x) {
  start.col = function(x){
    start.col = 20*x-16
    return(start.col)
  }
  sequence = mapply((start.col),c(x)):(mapply(start.col,c(x))+9)
  return(sequence)
}
PACK1 = DATA[,c(a(1),a(2),a(3),a(4),a(5),a(6),a(7),a(8),a(9),a(10))]
SPACK1 = scale(PACK1) # standardized data matrix
TRAINSET.PACK1 = rbind(SPACK1[1:s1,], SPACK1[(n1+1):(n1+s2),], SPACK1[(n1+n2+1):(n1+n2+s3),])
TESTSET.PACK1 = rbind(SPACK1[(s1+1):n1,], SPACK1[(n1+s2+1):(n1+n2),],
SPACK1[(n1+n2+s3+1):(n1+n2+n3),])

set.seed(1)
test.pred.kbest1 = knn(train = TESTSET.PACK1, test = TRAINSET.PACK1, cl = TESTSET_TARGET, k = kbest)
test.table.kbest1 = table(TRAINSET_TARGET, test.pred.kbest1)
print(test.table.kbest1)
w1 = sum(diag(test.table.kbest1))/sum(test.table.kbest1) # percentage of correct classifications by PACK1 on test set
print(w1)

# part 1.7
b = function(x) {
  start.col = function(x) {
    start.col = 20 * x - 6
    return(start.col)
  }
  sequence = mapply((start.col),c(x)):(mapply(start.col,c(x))+9)
  return(sequence)
}
d = function(x) {
  start.col = function(x) {
    start.col = 20 * x + 194
    return(start.col)
  }
  sequence = mapply((start.col),c(x)):(mapply(start.col,c(x))+9)
  return(sequence)
}
e = function(x) {
  start.col = function(x) {
    start.col = 20 * x + 184
    return(start.col)
  }
  sequence = mapply((start.col),c(x)):(mapply(start.col,c(x))+9)
  return(sequence)
}
PACK2 = DATA[,c(b(1),b(2),b(3),b(4),b(5),b(6),b(7),b(8),b(9),b(10))]
PACK3 = DATA[,c(d(1),d(2),d(3),d(4),d(5),d(6),d(7),d(8),d(9),d(10))]
PACK4 = DATA[,c(e(1),e(2),e(3),e(4),e(5),e(6),e(7),e(8),e(9),e(10))]

SPACK2 = scale(PACK2)
SPACK3 = scale(PACK3)
SPACK4 = scale(PACK4)

TRAINSET.PACK2 = rbind(SPACK2[1:s1,], SPACK2[(n1+1):(n1+s2),], SPACK2[(n1+n2+1):(n1+n2+s3),])
TRAINSET.PACK3 = rbind(SPACK3[1:s1,], SPACK3[(n1+1):(n1+s2),], SPACK3[(n1+n2+1):(n1+n2+s3),])
TRAINSET.PACK4 = rbind(SPACK4[1:s1,], SPACK4[(n1+1):(n1+s2),], SPACK4[(n1+n2+1):(n1+n2+s3),])

TESTSET.PACK2 = rbind(SPACK2[(s1+1):n1,], SPACK2[(n1+s2+1):(n1+n2),],
SPACK2[(n1+n2+s3+1):(n1+n2+n3),])
TESTSET.PACK3 = rbind(SPACK3[(s1+1):n1,], SPACK3[(n1+s2+1):(n1+n2),],
SPACK3[(n1+n2+s3+1):(n1+n2+n3),])
TESTSET.PACK4 = rbind(SPACK4[(s1+1):n1,], SPACK4[(n1+s2+1):(n1+n2),],
SPACK4[(n1+n2+s3+1):(n1+n2+n3),])

set.seed(1)
test.pred.kbest2 = knn(train = TESTSET.PACK2, test = TRAINSET.PACK2, cl = TESTSET_TARGET, k = kbest)
test.table.kbest2 = table(TRAINSET_TARGET, test.pred.kbest2)
w2 = sum(diag(test.table.kbest2))/sum(test.table.kbest2) # percentage of correct classifications by PACK2 on test set
print(w2)

test.pred.kbest3 = knn(train = TESTSET.PACK3, test = TRAINSET.PACK3, cl = TESTSET_TARGET, k = kbest)
test.table.kbest3 = table(TRAINSET_TARGET, test.pred.kbest3)
w3 = sum(diag(test.table.kbest3))/sum(test.table.kbest3) # percentage of correct classifications by PACK3 on test set
print(w3)
```

```
test.pred.kbest4 = knn(train = TESTSET.PACK4, test = TRAINSET.PACK4, cl = TESTSET_TARGET, k = kbest)
test.table.kbest4 = table(TRAINSET_TARGET, test.pred.kbest4)
w4 = sum(diag(test.table.kbest4))/sum(test.table.kbest4) # percentage of correct classifications by PACK4 on test set
print(w4)
```

# part 1.8

```
normalizer = 1/sum(w1,w2,w3,w4)
weight = function(x) {
  x*normalizer
}

W1 = PACK1/sum(PACK1)*weight(w1)
W2 = PACK2/sum(PACK2)*weight(w2)
W3 = PACK3/sum(PACK3)*weight(w3)
W4 = PACK4/sum(PACK4)*weight(w4)
sum(W1,W2,W3,W4) # to verify if their sum is 1 after renormalizing these 400 weights

W = cbind(W1,W2,W3,W4)
TRAINSET.W = rbind(W[1:s1,], W[(n1+1):(n1+s2),], W[(n1+n2+1):(n1+n2+s3),])
TESTSET.W = rbind(W[(s1+1):n1,], W[(n1+s2+1):(n1+n2),], W[(n1+n2+s3+1):(n1+n2+n3),])
set.seed(1)
train.pred.kbest.W = knn(train = TRAINSET.W, test = TESTSET.W, cl = TRAINSET_TARGET, k = kbest)
test.pred.kbest.W = knn(train = TESTSET.W, test = TRAINSET.W, cl = TESTSET_TARGET, k = kbest)

train.table.kbest.W = table(TESTSET_TARGET, train.pred.kbest.W)
test.table.kbest.W = table(TRAINSET_TARGET, test.pred.kbest.W)

trainperf.kbest.W = sum(diag(train.table.kbest.W))/sum(train.table.kbest.W)
testperf.kbest.W = sum(diag(test.table.kbest.W))/sum(test.table.kbest.W)

trainconf.kbest.W = prop.table(train.table.kbest.W, margin  = 1) # confusion matrix for training set
testconf.kbest.W = prop.table(test.table.kbest.W, margin  = 1) # confusion matrix for test set
```