k-NN Implementation

Vicent Caselles Ballester

2023-10-16

Contents

k-NN application to the dataset wisc__bc__data.csv on the variable diagnosis.

Pros and cons of k-NN algorithm

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

Table 1: **Strenghts and weaknesses of the k-NN algorithm**

Strengths	**Weaknesses**
Simple and effective	Does not produce a model, limiting the ability to understand how the features are related to the class
Makes no assumptions about the underlying data distribution	Requires selection of an appropriate k
Fast training phase	Slow classification phase
	Nominal features and missing data require additional processing

Data loading and first exploration

First of all, we upload the dataset.

```r
dataset = read.csv(file, stringsAsFactors = T)
```

We explore the dataset using the str function.

```r
str(dataset)
```

```
## 'data.frame':    569 obs. of  32 variables:
##  $ id               : int  87139402 8910251 905520 868871 9012568 906539 925291 87880 862989 89827 .
##  $ diagnosis        : Factor w/ 2 levels "B","M": 1 1 1 1 1 1 1 2 1 1 ...
##  $ radius_mean      : num  12.3 10.6 11 11.3 15.2 ...
##  $ texture_mean     : num  12.4 18.9 16.8 13.4 13.2 ...
##  $ perimeter_mean   : num  78.8 69.3 70.9 73 97.7 ...
##  $ area_mean        : num  464 346 373 385 712 ...
##  $ smoothness_mean  : num  0.1028 0.0969 0.1077 0.1164 0.0796 ...
##  $ compactness_mean : num  0.0698 0.1147 0.078 0.1136 0.0693 ...
##  $ concavity_mean   : num  0.0399 0.0639 0.0305 0.0464 0.0339 ...
##  $ points_mean      : num  0.037 0.0264 0.0248 0.048 0.0266 ...
##  $ symmetry_mean    : num  0.196 0.192 0.171 0.177 0.172 ...
##  $ dimension_mean   : num  0.0595 0.0649 0.0634 0.0607 0.0554 ...
##  $ radius_se        : num  0.236 0.451 0.197 0.338 0.178 ...
##  $ texture_se       : num  0.666 1.197 1.387 1.343 0.412 ...
##  $ perimeter_se     : num  1.67 3.43 1.34 1.85 1.34 ...
##  $ area_se          : num  17.4 27.1 13.5 26.3 17.7 ...
##  $ smoothness_se    : num  0.00805 0.00747 0.00516 0.01127 0.00501 ...
##  $ compactness_se   : num  0.0118 0.03581 0.00936 0.03498 0.01485 ...
##  $ concavity_se     : num  0.0168 0.0335 0.0106 0.0219 0.0155 ...
##  $ points_se        : num  0.01241 0.01365 0.00748 0.01965 0.00915 ...
##  $ symmetry_se      : num  0.0192 0.035 0.0172 0.0158 0.0165 ...
##  $ dimension_se     : num  0.00225 0.00332 0.0022 0.00344 0.00177 ...
##  $ radius_worst     : num  13.5 11.9 12.4 11.9 16.2 ...
##  $ texture_worst    : num  15.6 22.9 26.4 15.8 15.7 ...
##  $ perimeter_worst  : num  87 78.3 79.9 76.5 104.5 ...
##  $ area_worst       : num  549 425 471 434 819 ...
##  $ smoothness_worst : num  0.139 0.121 0.137 0.137 0.113 ...
##  $ compactness_worst: num  0.127 0.252 0.148 0.182 0.174 ...
##  $ concavity_worst  : num  0.1242 0.1916 0.1067 0.0867 0.1362 ...
##  $ points_worst     : num  0.0939 0.0793 0.0743 0.0861 0.0818 ...
##  $ symmetry_worst   : num  0.283 0.294 0.3 0.21 0.249 ...
##  $ dimension_worst  : num  0.0677 0.0759 0.0788 0.0678 0.0677 ...
```

We drop the id variable, which we won't be needing.

```r
if ("id" %in% colnames(dataset)){
  dataset <- dataset[-which(colnames(dataset)=='id')]
}
```

We check how many of the tumors are benign (B) and how many are malignant (M).

```r
table(dataset$diagnosis)
```

```
##
##   B   M
## 357 212
```

We recode the variable of interest to a factor.

```r
if (!is.factor(dataset[[variable_of_interest]])) {
  dataset[[variable_of_interest]] <- as.factor(dataset[[variable_of_interest]])
}
```

```
round(prop.table(table(dataset[[variable_of_interest]])) * 100, digits = 1)
```

```
##
##    B    M
## 62.7 37.3
```

Min/max normalization

We normalize all features and substract the label

```
dataset_normalized = as.data.frame(lapply(dataset[, -which(colnames(dataset) == variable_of_interest)],
                                          normalize))
```

We check everything worked out A-OK.

```
summary(dataset_normalized$area_mean)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.1174  0.1729  0.2169  0.2711  1.0000
```

We define the function that I'll use to generate the splits for training/test.

```
get_splits_percentage <- function(n_dataset){
  if (n_dataset >= 500){
    n_test = 100
    n_train = n_dataset - 100
  }
  else {
    n_train = ceiling(n_dataset * 0.8) # random split decided by moi
    n_test = floor(n_dataset * 0.2)
  }
  return(c(n_train, n_test))
}
```

Fitting the model

We define the dataset to used for training and testing.

```
splits_to_use=get_splits_percentage(nrow(dataset_normalized))
dataset_train = dataset_normalized[1:splits_to_use[1], ]
dataset_test = dataset_normalized[(splits_to_use[1]+1):nrow(dataset_normalized), ]
labels_train = dataset[[variable_of_interest]][1:splits_to_use[1]]
labels_test = dataset[[variable_of_interest]][(splits_to_use[1]+1):nrow(dataset_normalized)]
```

Now we'll use the `knn` function from the `class` package to fit the model.

```
require(class)
```

```
## Loading required package: class
```

```
test_pred = knn(train = dataset_train, test = dataset_test,
                cl = labels_train, k = 21)
```

```
require(gmodels)
```

```
## Loading required package: gmodels
```

```r
CrossTable(x = labels_test, y=test_pred, prop.chisq = F)
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  100
##
##
##              | test_pred
##  labels_test |         B |         M | Row Total |
## -------------|-----------|-----------|-----------|
##            B |        61 |         0 |        61 |
##              |     1.000 |     0.000 |     0.610 |
##              |     0.968 |     0.000 |           |
##              |     0.610 |     0.000 |           |
## -------------|-----------|-----------|-----------|
##            M |         2 |        37 |        39 |
##              |     0.051 |     0.949 |     0.390 |
##              |     0.032 |     1.000 |           |
##              |     0.020 |     0.370 |           |
## -------------|-----------|-----------|-----------|
## Column Total |        63 |        37 |       100 |
##              |     0.630 |     0.370 |           |
## -------------|-----------|-----------|-----------|
##
##
```

Improving the model

Z-score Transformation

Now we try with z-score normalization.

```r
dataset_zscored = as.data.frame(scale(dataset[, -which(colnames(dataset) == variable_of_interest)]))
summary(dataset_zscored$area_mean)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -1.4532 -0.6666 -0.2949  0.0000  0.3632  5.2459
```

We define the training and test set again.

```r
train_zscore = dataset_zscored[1:splits_to_use[1], ]
test_zscore = dataset_zscored[(splits_to_use[1]+1):nrow(dataset_zscored), ]
```

We predict again the labels now using the zscore normalized dataset, and visualize the results using the CrossTable function.

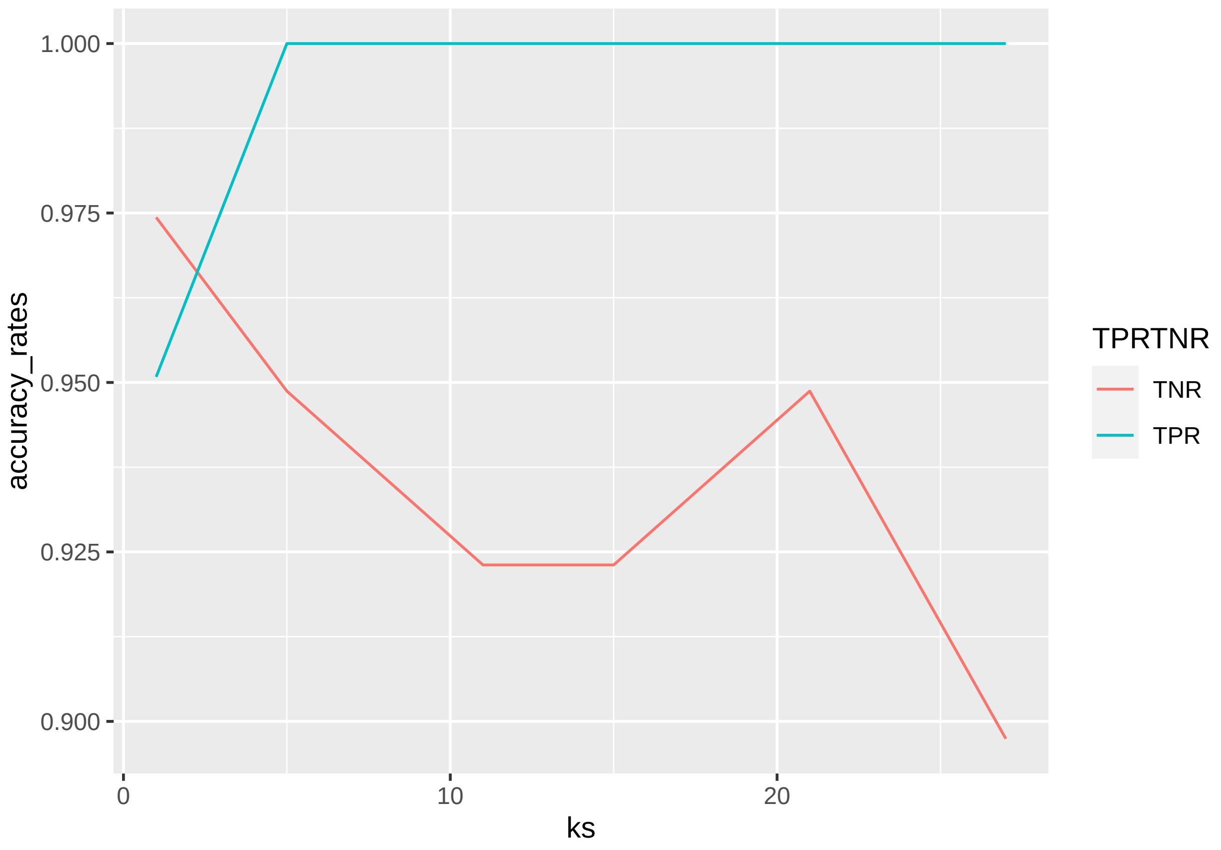
```r
zscore_pred_test = knn(train=train_zscore, test=test_zscore,
                       cl=labels_train, k = ceiling(sqrt(nrow(train_zscore))))
```

```r
CrossTable(x=labels_test, y=zscore_pred_test, prop.chisq=F)
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  100
##
##
##              | zscore_pred_test
##  labels_test |         B |         M | Row Total |
## -------------|-----------|-----------|-----------|
##            B |        61 |         0 |        61 |
##              |     1.000 |     0.000 |     0.610 |
##              |     0.924 |     0.000 |           |
##              |     0.610 |     0.000 |           |
## -------------|-----------|-----------|-----------|
##            M |         5 |        34 |        39 |
##              |     0.128 |     0.872 |     0.390 |
##              |     0.076 |     1.000 |           |
##              |     0.050 |     0.340 |           |
## -------------|-----------|-----------|-----------|
## Column Total |        66 |        34 |       100 |
##              |     0.660 |     0.340 |           |
## -------------|-----------|-----------|-----------|
##
##
```

Alternative values of k

I'm not showing the code used to generate the data because it's ugly. If you wanna know my secrets open the Rmd file directly (please don't).



Appendix: "(Super-slow) Manual implementation of k-NN algorithm"

Just for fun I'm gonna implement the k-NN algorithm myself.

```r
get_distance_obs <- function(vector1, vector2){
  return(sqrt(sum((vector1 - vector2)**2)))
}

compute_distance_y_to_train_observations <- function(y, train_obs){
  distances = c()
  for (row in 1:nrow(train_obs)){
    distances = append(distances, get_distance_obs(y, train_obs[row, ]))
  }
  return(distances)
}

subset_k_distances <- function(vector_distances, k){
  # Returns indices of values that minimize our distance function
  k_min_distances = which(vector_distances <= max(sort(vector_distances, decreasing = F)[1:k]))
  return(k_min_distances)
}

get_labels_corresponding_to_k_neighbors <- function(train_labels, indices){
  return(train_labels[indices])
}
```

```r
predict_one_observation <- function(y_obs, train_data, k, train_labels){
  distances_to_train_data <- compute_distance_y_to_train_observations(y_obs, train_data)
  k_distance_indices = subset_k_distances(distances_to_train_data, k=k)
  labels_sliced = get_labels_corresponding_to_k_neighbors(train_labels, k_distance_indices)
  return(names(sort(summary(labels_sliced), decreasing = T)[1]))
}

predict_all_observations <- function(dataset_with_ys, train_data, k, train_labels){
  predicted_labels_vec = c()
  if (is.data.frame(dataset_with_ys)){
    n_observations_to_predict = nrow(dataset_with_ys)
    for (obs in 1:n_observations_to_predict){
      predicted_label = predict_one_observation(y_obs=dataset_with_ys[obs, ],
                                                train_data = train_data,
                                                k=k, train_labels = train_labels)
      predicted_labels_vec = append(predicted_labels_vec, predicted_label)
    }
  }
  else {
    stop("Non supported data type")
  }
  return(predicted_labels_vec)
}
```

```r
labels_by_moi = predict_all_observations(dataset_with_ys = dataset_test, train_data = dataset_train,
                                         k=21, train_labels = labels_train)
using_knn_already_implemented = knn(train=dataset_train, test=dataset_test, cl=labels_train, k=21)

labels_by_moi == using_knn_already_implemented
```

```
##   [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [16] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [31] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [46] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [61] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [76] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [91] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Com veiem, em dóna igual (aleluya). El meu codi és probablement 10000000 vegades més lent que la funció knn del paquet `class`, però ara mateix no tinc temps per a millorar-ho (i crec que tampoc és l'objectiu de l'activitat). Ho faria utilitzant apply enlloc de utilitzar `for loops`, i si fos en `Python` faria servir numpy i vectorització, no sé si a R hi ha algun equivalent.