Prediction of E. Coli promoter gene sequences

Rubén Sánchez Fernández

30, octubre, 2018

Contents

Introduction

This report aims to introduce one of the simplest machine learning algorithms, the k-Nearest Neighbors. k-NN is a supervised, non-parametric method used both for classification and regression. It's simplicity and efectiveness makes it one of the widest used ML algorithms.

K-NN algorithm

K-NN method is based on distance measure. Each unlabeled point is classified according to which class has the highest frequency from the *k* nearest points. When performing regression, the output is the mean or median from the *k* nearest points.

(Lantz 2015) summarized the strengths and weaknesses of this 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 appropiate *k*
· Fast training phase	· Slow classification phase
-	· Nominal features and missing data require additional processing

As previously mentioned, k-NN measures 'similarity' by calculating the distance between points. There are several distance measures that can be implemented with k-NN, being **Euclidean distance** the more popular.

Eucliden distance is calculated following the next equation:

$dist(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_n - q_n)^2}$

Where p and q are the two samples and n is the feature. Therefore, p_1 is the point of the sample p on the first feature, and p_n is the point of sample p on the last feature.

Example: E.coli promoter gene sequence prediction with k-NN

Let's use the k-NN algorithm to predict if a gene sequence is a promoter sequence of *Escherichia coli* (Harley and Reynolds 1987).

We will solve this exercise following a 5 step process:

- **Step 1**: Collecting data
- **Step 2**: Exploring and preparing the data
- **Step 3**: Training a model on the data
- **Step 4**: Evaluating model performance
- **Step 5**: Improving model performance

Step 1: Collecting data

We will use the "Molecular Biology (Promoter Gene Sequences) Data Set" which can be downloaded from the UCI Machine Learning repository. The file downloaded is a *csv* file that can be easily imported to R using the *read.csv()* function.

```r
#importing data
data<-read.csv(file1, header = FALSE)
```

Step 2: Exploring and preparing the data

```r
#internal structure of dataset
str(data)
```

```
## 'data.frame':    106 obs. of  3 variables:
##  $ V1: Factor w/ 2 levels "-","+": 2 2 2 2 2 2 2 2 2 2 ...
##  $ V2: Factor w/ 106 levels "1019","1024",..: 92 55 58 62 71 75 76 80 83 84 ...
##  $ V3: Factor w/ 106 levels "aaaaacgtcatcgcttgcattagaaaggtttctggccgaccttataaccattaatta",..: 71 88 67
```

The dataset is structured as a dataframe of 106 observations and 3 variables. The first variable is the response variable, indicating if the sequence is promoter (+) or not (-). The second variable is the sequence ID and the third variable is the sequence. For convenience, let's name each variable:

```r
#naming the variables
colnames(data) <- c("promoter","name","sequence")
#show first rows of the dataset
head(data)
```

```
##   promoter      name
## 1        +       S10
## 2        +      AMPC
## 3        +      AROH
## 4        +     DEOP2
## 5        + LEU1_TRNA
## 6        +    MALEFG
##                                                      sequence
## 1 tactagcaatacgcttgcgttcggtggttaagtatgtataatgcgcgggcttgtcgt
## 2 tgctatcctgacagttgtcacgctgattggtgtcgttacaatctaacgcatcgccaa
## 3 gtactagagaactagtgcattagcttatttttttgttatcatgctaaccacccggcg
## 4 aattgtgatgtgtatcgaagtgtgttgcggagtagatgttagaatactaacaaactc
## 5 tcgataattaactattgacgaaaagctgaaaaccactagaatgcgcctccgtggtag
## 6 aggggcaaggaggatggaaagaggttgccgtataaagaaactagagtccgtttaggt
```

One-hot encoding

Only the *sequence* variable will be used to train the classification model. But we can't use the raw sequence to train the model, first we need to convert each nucleotide (A,T,C,G) inside the sequences to a number. In this case, the best approach is to use *one-hot encoding*, where each nucleotide is transformed to a 4-bit vector. The transformation goes as:

$A = [1, 0, 0, 0]$

$C = [0, 1, 0, 0]$

$G = [0, 0, 1, 0]$

$T = [0, 0, 0, 1]$

```r
# ONE-HOT ENCODING FUNCTION FOR GENOMIC SEQUENCES
# Author: Rubén Sánchez Fernández

one_hot <- function(data, column_name){

  #creating an empty list to save the encoded sequences
  ls <- list()
  #going through each sequence
  for (n_seq in 1:nrow(data)) {

    #converting to string
    x <- as.character(data[n_seq, column_name])

    #accessing to each character inside the sequence
    x_split <- strsplit(x, "")[[1]]

    #sequence length
    n <- length(nchar(x_split))

    #empty matrix
    m1 <- matrix(nrow = n, ncol = 4)

    #going through each character and assigning the binary vectors
    for (i in 1:n){
      if (x_split[i] == "a"){
        m1[i,] = c(1,0,0,0)}
      if (x_split[i] == "c"){
        m1[i,] = c(0,1,0,0)}
      if (x_split[i] == "g"){
        m1[i,] = c(0,0,1,0)}
      if(x_split[i] == "t"){
        m1[i,] = c(0,0,0,1)}
    }
    #adding each encoded sequence into the list
    ls[[n_seq]] <- m1
  }

  #we're going to save the sequences also as a dataframe to input them in the ML algorithm
  df <- matrix(nrow = length(ls), ncol = n*4)

  for (i in 1:length(ls)){
    df[i,] <- as.vector(t(ls[[i]])) #transposing and converting to vector each sequence
  }
  df <- as.data.frame(df) #coverting to df

  #storing list with enc. sequences and dataframe with enc.sequences
  results <- list(ls, df)
  names(results) <- c("List encoded sequences", "Dataframe encoded sequences")

  return(results)}
```

We created a function to perform one-hot encoding. Now, we just have to call it.

```
#encoding dna sequences
enc_seq <- one_hot(data, "sequence")
```

Let's check the first sequence.

```
enc_seq$`List encoded sequences`[[1]]
```

```
##       [,1] [,2] [,3] [,4]
##  [1,]    0    0    0    1
##  [2,]    1    0    0    0
##  [3,]    0    1    0    0
##  [4,]    0    0    0    1
##  [5,]    1    0    0    0
##  [6,]    0    0    1    0
##  [7,]    0    1    0    0
##  [8,]    1    0    0    0
##  [9,]    1    0    0    0
## [10,]    0    0    0    1
## [11,]    1    0    0    0
## [12,]    0    1    0    0
## [13,]    0    0    1    0
## [14,]    0    1    0    0
## [15,]    0    0    0    1
## [16,]    0    0    0    1
## [17,]    0    0    1    0
## [18,]    0    1    0    0
## [19,]    0    0    1    0
## [20,]    0    0    0    1
## [21,]    0    0    0    1
## [22,]    0    1    0    0
## [23,]    0    0    1    0
## [24,]    0    0    1    0
## [25,]    0    0    0    1
## [26,]    0    0    1    0
## [27,]    0    0    1    0
## [28,]    0    0    0    1
## [29,]    0    0    0    1
## [30,]    1    0    0    0
## [31,]    1    0    0    0
## [32,]    0    0    1    0
## [33,]    0    0    0    1
## [34,]    1    0    0    0
## [35,]    0    0    0    1
## [36,]    0    0    1    0
## [37,]    0    0    0    1
## [38,]    1    0    0    0
## [39,]    0    0    0    1
## [40,]    1    0    0    0
## [41,]    1    0    0    0
## [42,]    0    0    0    1
## [43,]    0    0    1    0
## [44,]    0    1    0    0
## [45,]    0    0    1    0
## [46,]    0    1    0    0
## [47,]    0    0    1    0
## [48,]    0    0    1    0
## [49,]    0    0    1    0
## [50,]    0    1    0    0
## [51,]    0    0    0    1
## [52,]    0    0    0    1
## [53,]    0    0    1    0
## [54,]    0    0    0    1
## [55,]    0    1    0    0
## [56,]    0    0    1    0
## [57,]    0    0    0    1
```

Creating training and test subsets

Now, we have to split the data in two subsets: training and test. The training set will have the 67% of the data, random separated, and the rest will go to the test set.

```
#We will work with the encoded sequences saved as [106,228]: 1 full sequence per row
#(knn function requires data to be in matrix or data frame form)
enc_seq_df <- enc_seq$`Dataframe encoded sequences`

#randomly separating data into training and test sets
# 67% training
size <- floor(0.67 * nrow(data))

#set seed to assure reproducibility
set.seed(123)

#training indices
trn <- sample(seq_len(nrow(data)), size = size)

#training sequences
train <- enc_seq$`Dataframe encoded sequences`[trn,]

#training labels
train_labels <- data[trn,1]

#test sequences
test <- enc_seq$`Dataframe encoded sequences`[-trn,]

#test labels
test_labels <- data[-trn,1]
```

Step 3: Training a model on the data

K-NN is what we call a *lazy learner*, training phase only consists in storing the input data in a structured format, which is already done. Now, we can classify the test data using the *knn()* function from the *class* package. In this case, we will start using k = 3.

```
#k = 3
prediction<-knn(train=train, test=test, cl=train_labels, k=3)
summary(prediction)
```

```
##  -  +
## 11 24
```

We obtained a vector with the predicted labels. In the next step, we will asess how true are this predictions and therefore, how accurate is the model.

Step 4: Evaluating model performance

With the *confusionMatrix()* function of the *caret* package we can obtain the confusion matrix and also other performance measures with just one line of code.

```
confusionMatrix(prediction, test_labels, positive = "+")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  -  +
##          - 10  1
##          +  9 15
##
##                Accuracy : 0.7143
##                  95% CI : (0.537, 0.8536)
##     No Information Rate : 0.5429
##     P-Value [Acc > NIR] : 0.02934
##
##                   Kappa : 0.4462
##  Mcnemar's Test P-Value : 0.02686
##
##             Sensitivity : 0.9375
##             Specificity : 0.5263
##          Pos Pred Value : 0.6250
##          Neg Pred Value : 0.9091
##              Prevalence : 0.4571
##          Detection Rate : 0.4286
##    Detection Prevalence : 0.6857
##       Balanced Accuracy : 0.7319
##
##        'Positive' Class : +
##
```

The number of false positive (FP) is 9 and the the number of false negative (FN) is 1. The accuracy of this model is 0.7143, therefore the error rate is 0.2857.

Step 5: Improving model performance

We will try to improve the model performance by changing the k value.

```
#k = 5
prediction<-knn(train=train, test=test, cl=train_labels, k=5)
#confusion matrix
confusionMatrix(prediction, test_labels, positive = "+")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  -  +
##          -  6  1
##          + 13 15
##
##                Accuracy : 0.6
##                  95% CI : (0.4211, 0.7613)
##     No Information Rate : 0.5429
##     P-Value [Acc > NIR] : 0.307007
##
##                   Kappa : 0.2391
##  Mcnemar's Test P-Value : 0.003283
##
##             Sensitivity : 0.9375
##             Specificity : 0.3158
##          Pos Pred Value : 0.5357
##          Neg Pred Value : 0.8571
##              Prevalence : 0.4571
##          Detection Rate : 0.4286
##    Detection Prevalence : 0.8000
##       Balanced Accuracy : 0.6266
##
##        'Positive' Class : +
##
```

```
#k = 7
prediction<-knn(train=train, test=test, cl=train_labels, k=7)
#confusion matrix
confusionMatrix(prediction, test_labels, positive = "+")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  -  +
##          -  8  1
##          + 11 15
##
##                Accuracy : 0.6571
##                  95% CI : (0.4779, 0.8087)
##     No Information Rate : 0.5429
##     P-Value [Acc > NIR] : 0.116877
##
##                   Kappa : 0.3417
##  Mcnemar's Test P-Value : 0.009375
```

```
##
##             Sensitivity : 0.9375
##             Specificity : 0.4211
##          Pos Pred Value : 0.5769
##          Neg Pred Value : 0.8889
##              Prevalence : 0.4571
##          Detection Rate : 0.4286
##    Detection Prevalence : 0.7429
##       Balanced Accuracy : 0.6793
##
##        'Positive' Class : +
##
```

```
#k=11
prediction<-knn(train=train, test=test, cl=train_labels, k=11)
#confusion matrix
confusionMatrix(prediction, test_labels, positive = "+")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  -  +
##          -  8  0
##          + 11 16
##
##                Accuracy : 0.6857
##                  95% CI : (0.5071, 0.8315)
##     No Information Rate : 0.5429
##     P-Value [Acc > NIR] : 0.061949
##
##                   Kappa : 0.3994
##  Mcnemar's Test P-Value : 0.002569
##
##             Sensitivity : 1.0000
##             Specificity : 0.4211
##          Pos Pred Value : 0.5926
##          Neg Pred Value : 1.0000
##              Prevalence : 0.4571
##          Detection Rate : 0.4571
##    Detection Prevalence : 0.7714
##       Balanced Accuracy : 0.7105
##
##        'Positive' Class : +
##
```

The results are summarized in the next table:

k value	False negatives	False positives	Percent classified incorrectly
3	1	9	28.57%
5	1	13	40%
7	1	11	34.29%
11	0	11	31.43%

The best performance is obtained with k = 3 and k = 11. With k = 3, we managed to get the lowest error rate, 28.57%. With k = 11, we got 31.43% of error rate, but 0 false negatives. Choosing between these two models would depend on how important is sensitivity for us. If we want the model with the highest sensitivity possible, we would choose k = 11, in which we obtained a 100% of sensitivity. If having perfect sensitivity is not important for us, we would choose k = 3, in which we obtained worst sensitivity (93.75%) but better specificity and better classification performance overall.

Bibiliography

Harley, Calvin B, and Robert P Reynolds. 1987. "Analysis of E. Coli Pormoter Sequences." *Nucleic Acids Research* 15 (5). Oxford University Press: 2343–61.

Lantz, Brett. 2015. *Machine Learning with R*. Packt Publishing Ltd.