Lecture notes on Bioinformatics

Madina Japakhova

August 2022

To see this html page rendered you can use: https://htmlpreview.github.io/

Readme

Lecture notes on "Applied Bioinformatics" (Summer '22, TU Dresden, Prof. Michael Schroeder). The implementation of algorithms taught in the course was shown in Python. Following the python code provided by Prof. Michael Schroeder and his team, I rewrote the shown algorithms in **R** and solved some of the homework tasks.

Levenshtein distance (or edit distance)

is the minimum number of edit operations (**insertion, deletion, substitution**) that are needed to convert one string into the other. The Levenshtein distance allows to compare and align any two strings. The algorithm can be applied in spell checking and speech recognition. In computational biology and bioinformatics, the edit distance is used to measure differences and similarities in sequences (nucleotides, amino acids).

The implementation technique for the algorithm is **dynamic programming**. Important: the written script is case sensitive.

```
lev <- function(a,b){
  # initialise dynamic programming matrix
  d = matrix(rep(0, (nchar(a)+1)*(nchar(b)+1)),nrow = nchar(a)+1, ncol = nchar(b)+1)

  # initialise first column
  for (i in 1:(nchar(a)+1)){
    d[i,1] = i-1
  }

  # initialise first row
  for (j in 1:(nchar(b)+1)){
    d[1,j] = j-1
  }

  # fill in rest of the matrix
  for (i in 2:(nchar(a)+1)){
    for (j in 2:(nchar(b)+1)){
      # if letters are the same, no edit operation is used,
      # otherwise it is one (replace operation)
      if (substr(a,i-1,i-1) == substr(b,j-1, j-1)){
        s = 0
      } else {
        s = 1
      }
      # Get the minimum of the scores
      d[i,j] = min(d[i-1,j]+1, d[i,j-1]+1, d[i-1,j-1] + s)
      }}
  return(d[nchar(a)+1,nchar(b)+1])
}

a <- "Christmas"
d <- "Fantasy"
lev(a,d)
```

```
## [1] 7
```

The minimum number of operations to convert string “Christmas” into “Fantasy” is 7, that is the edit distance is 7. This is the correct answer.
Alternatively, to calculate the edit distance between two strings you can use a one-liner from library *stringdist* :D

```
library(stringdist)
#calculate Levenshtein distance between Christmas and Fantasy
stringdist("Christmas", "Fantasy", method = "lv")
```

```
## [1] 7
```

HW PubMed task: implementation of Levenshtein distance in Pubmed search

Read a file with names of authors on PubMed. Use Levenshtein distance to return names similar to the query, that is given a query name, return all other names with an edit distance smaller than a selected threshold. Exclude same names from the search result.

```
library(stringr)
library(magrittr)
```

```
##
## Attaching package: 'magrittr'

## The following object is masked from 'package:stringdist':
##
##     extract
```

```
library(readr)
pubmed <- read.delim("Pubmed.Forename.txt", sep = " ")

# limit search to first names only
pubmed$first_name <- str_extract(pubmed$Fore, '[A-Za-z]+')

# to fasten things a bit, subset to just 1000 cases (original data ~400 000 observations)
pubmed <- pubmed[1:1000,] %>%
  na.omit(pubmed)

similarity_threshold <- 2
similar_names <- data.frame(similar_name=character(),stringsAsFactors=FALSE)

similar <- function(author){

  for(i in 1:nrow(pubmed)){
    word <- pubmed[i,3]
    if(lev(author, word) <= similarity_threshold & lev(author, word) != 0){
      similar_names[i,1] <- word
      similar_names <- na.omit(similar_names)
    }
  }
  return(similar_names)

}
a <-similar(author = "Peter")

# write a .csv file with all the names that are less than two edit operations similar to "Peter"
write_csv(a, "Peter_similar_names_pubmed.csv")
```

Longest common subsequence (lcs)

Subsequence *vs* substring. A *substring* takes characters between two specified position indices (or a single character at a single position) from a string in a contiguous order. A *subsequence* is a generalization of a substring when contiguity is not important, however, relative order of characters as they appear in the query has to be maintained. Example, given string is “Elephant”, possible substrings are “e”, “el”, “ele”, “phan”, empty string. Possible subsequences include “eph”, lant“,”eh". An empty string is a substring of any string. The entire substring is a substring of itself.

Longest common subsequence algorithm return one common longest subsequence between two given sequences. LCS maximizes number of matches.

Levenshtein *vs.* LCS. It is easy to change the code from the Levenshtein distance to LCS:

1) LCS uses a reverted scoring scheme. Match is given a score of 1, mismatch, insertion, and deletion scores are all zeros.
2) An empty string is a substring of any string and in this case LCS is zero. Therefore, we have only zeros in the first row and column of the dynamic programming matrix.
3) LCS *maximizes* the number of matches, therefore we choose *max* instead of *min* of the 3 cells.

The script is case sensitive.

```
# from lev to lcs: 1) first row, first column in d initialization: all zeros
#                  2) reverted scoring scheme: +1 for matches, 0 for anything else
#                  3) max instead of min of the three cells
rm(list=ls())
lcs <- function(a,b){
  # initialise dynamic programming matrix
  d = matrix(rep(0, (nchar(a)+1)*(nchar(b)+1)),nrow = nchar(a)+1, ncol = nchar(b)+1)

  # initialise first column: 0. Empty string is a substring of any string, lcs = 0
  for (i in 1:(nchar(a)+1)){
    d[i,1] = 0
  }

  # initialise first row: 0. Empty string is a substring of any string, lcs = 0
  for (j in 1:(nchar(b)+1)){
    d[1,j] = 0
  }

  # fill in rest of the matrix
  for (i in 2:(nchar(a)+1)){
    for (j in 2:(nchar(b)+1)){
      # reward a match, zero for anything else (mismatch, insertion, deletion = 0)
      if (substr(a,i-1,i-1) == substr(b,j-1, j-1)){
        s = 1
      } else {
        s = 0
      }

      # Get the maximum of the scores
      d[i,j] = max(d[i-1,j], d[i,j-1], d[i-1,j-1] + s)
    }}
  return(d[nchar(a)+1,nchar(b)+1])
}


str1 <- "world"
str2 <- "leader"
lcs(str1, str2)
```

```
## [1] 2
```

The returned results is 2. It is correct (‘ld’)

Global alignment. Needleman-Wunsch algorithm.

The script is case sensitive.

```
rm(list=ls())

# Global alignment: Needleman-Wunsch algorithm (NW)

# from LCS to NW: 1) first row, first column in d initialization: accumulate (negative) gap penalties
#                 2) adapt scoring scheme

nw <- function(a,b, s_m, s_r, s_g){
  # initialise dynamic programming matrix
  d = matrix(rep(0, (nchar(a)+1)*(nchar(b)+1)),nrow = nchar(a)+1, ncol = nchar(b)+1)

  # accumulate gap penalties in the 1st query
  for (i in 1:(nchar(a)+1)){
    d[i,1] = (i-1)*s_g
  }

  # accumulate gap penalties in the 2nd query
  for (j in 1:(nchar(b)+1)){
    d[1,j] = (j-1)*s_g
  }

  # fill in rest of the matrix
  for (i in 2:(nchar(a)+1)){
    for (j in 2:(nchar(b)+1)){
      if (substr(a,i-1,i-1) == substr(b,j-1, j-1)){
        s = s_m
      } else {
        s = s_r
      }
      # Get the maximum of the scores
      d[i,j] = max(d[i-1,j] + s_g, d[i,j-1] + s_g, d[i-1,j-1] + s)
    }}
  return(d[nchar(a)+1,nchar(b)+1])
}


str1 <- "sweet"
str2 <- "sweat"
nw(str1, str2, 1, -1, -2)
```

```
## [1] 3
```

The returned result is 3. It is correct.