Trabajo Final Optimización Heurística

Profesores: Julián Arias y Danny Munera

Estudiante: Jonathan Medina Gómez

Facultad de Ingeniería

Universidad de Antioquia

25 de Abril de 2017

Medellín, Antioquia

Planteamiento del problema

En el artículo de Sun (2010)[1] plantean un algoritmo para una medida de similitud entre pacientes llamado “Locally Supervised Metric Learning”. El propósito de este algoritmo es aprender de una medida de distancia de Mahalanobis entre un paciente x_i y un paciente x_j, definida de la siguiente manera:

$$d_m(x_i,\ x_j)\ =\ \sqrt{(x_i - x_j)^T\ P\ (x_i - x_j)}$$

Donde P es llamada matriz de precisión, la cual debe ser semidefinida positiva y es usada para incorporar las correlaciones entre las diferentes características de los dos pacientes. La clave del algoritmo es aprender una matriz P óptima tal que la métrica de distancia tenga las siguientes propiedades:

- Compacidad dentro de la clase: los pacientes con la misma etiqueta deben estar cerca.
- Dispersión entre clases: pacientes con etiquetas diferentes deben quedar lejos entre sí.

Para cumplir con estas propiedades se plantean dos tipos de vecindarios:

- El vecindario homogéneo de x_i denotado por N_i^o, el cual está compuesto por los k pacientes más cercanos a x_i con la misma etiqueta.
- El vecindario heterogéneo: de x_i denotado por N_i^e, el cual está compuesto por los k pacientes más cercanos a x_i con diferente etiqueta.

De acuerdo a estos dos vecindarios, se puede definir la compacidad local de la siguiente manera:

$$C_i\ =\ \sum_{x_j \varepsilon\, N_i^o} d_m^2(x_i,\ x_j)$$

Y la dispersión local como:

$$S_i\ =\ \sum_{x_k \varepsilon\, N_i^e} d_m^2(x_i,\ x_k)$$

La discriminabilidad de la medida de distancia d_m es definida como:

[1] "Localized Supervised Metric Learning on Temporal Physiological Data." https://pdfs.semanticscholar.org/fcf6/b5d6859244daab74a14ad9b89b0e9f0fbd66.pdf. Se consultó el 25 abr.. 2017.

$$\gamma = \frac{\sum_i C_i}{\sum_i S_i} = \frac{\sum_i \sum_{x_j \varepsilon N_i^o} (x_i - x_j)^T P (x_i - x_j)}{\sum_i \sum_{x_k \varepsilon N_i^e} (x_i - x_k)^T P (x_i - x_k)}$$

El objetivo es encontrar una matriz P que haga mínimo a γ, lo cual es equivalente a minimizar la compacidad local y maximizar la dispersión local simultáneamente.
En el artículo de Sun (2010) realizan esta optimización con el método descompuesto de Newton, en este trabajo se pretende realizar la optimización por medio de las técnicas heurísticas Simulated Annealing y algoritmos genéticos.

Conjunto de datos

Para este trabajo se utilizó una base de datos de acceso libre llamada MIMIC-III, la cual es desarrollada por un laboratorio del MIT y contiene datos de admisiones de pacientes en la unidad de cuidados intensivos (UCI) del hospital Beth Israel Deaconess entre los años 2001 y 2012. Se filtraron lo pacientes de interés para el problema que se está tratando en el proyecto de maestría y quedaron 1000 pacientes con 80 variables cada uno. Cada paciente tiene una etiqueta que toma el valor de “1” si el paciente no logró sobrevivir más de un año después de ingresar a la UCI y “0” si logra sobrevivir más del año.

Procedimiento

Las simulaciones se realizaron en el software estadístico R. Lo primero que se desarrolló fue la función para obtener los datos del archivo CSV y obtener una primera matriz P aleatoria para iniciar las simulaciones. Los números aleatorios se generaron con una distribución uniforme desde 0 hasta 1, ya que la matriz P representa las correlaciones entre las características de los pacientes:

```r
#return a vector with the upper triangle matrix elements of P matrix
getPSample <- function(){

  #initialization of variables
  detP = 0
  nFeatures = ncol(dataInput)-1

  #creation of a symetric matrix with det>0
  while (detP <= 0){
    #creating a square matrix with random values between 0, 1
    pMatrix = matrix(runif(nFeatures^2, min = 0, max = 1), nrow = nFeatures, ncol = nFeatures)
    #setting symetric matrix
    pMatrix[lower.tri(pMatrix)] = t(pMatrix)[lower.tri(pMatrix)]
    #setting 1 in main diagonal
    diag(pMatrix) = 1
    #validating determinant > 0
    detP = det(pMatrix)
  }
  #getting the upper triangle matrix from P matrix
  pMatrix[lower.tri(pMatrix, diag = TRUE)] = NA
  pSample = as.vector(t(pMatrix))
  pSample = pSample[!is.na(pSample)]
  return(pSample)
}
```

Imagen 1. Creación de las muestras para las heurísticas

Luego de esta matriz se extrajo la matriz triangular superior y se convirtió a un vector, el cual será utilizado como muestra en las heurísticas.

```r
#getting the upper triangle matrix from P matrix
pMatrix[lower.tri(pMatrix, diag = TRUE)] = NA
pSample = as.vector(t(pMatrix))
pSample = pSample[!is.na(pSample)]
return(pSample)
```

Imagen 2. Extracción de los datos de la matriz triangular superior.

Luego se implementó la función para evaluar la calidad de una solución sobre el problema planteado. En esta, se definió el tamaño de los vecindarios, luego se procedió a recorrer la base de datos y se obtuvieron los resultados de la medida de distancia de Mahalanobis de cada paciente con el resto. Posteriormente se calculó la compacidad y la dispersión local y en última instancia se obtuvo la discriminabilidad que es la función que se desea minimizar.

```
objectiveFunction <- function(solution){

  #transforming sample in P matrix
  pMatrix = matrix(1L, nrow = 80, ncol = 80, byrow = TRUE)
  pMatrix[lower.tri(pMatrix, diag = FALSE)] <- solution
  pMatrix = t(pMatrix)
  pMatrix[lower.tri(pMatrix)] = t(pMatrix)[lower.tri(pMatrix)]
  x = c(2:ncol(dataInput))

  # parameter: length of neighborhoods
  k = 5

  #Neighborhoods
  compactNeighborhood = matrix(0L, nrow = nrow(pMatrix))
  scatterNeighborhood = matrix(0L, nrow = nrow(pMatrix))
  for (i in 1:1000){
    neighbour =1
    xi = as.numeric(dataInput[i,x])
    generalMahalanobis = matrix(data = 0L, nrow = nrow(dataInput), ncol = 2)
    for (j in 1:1000){
      if (i != j){
        xj = as.numeric(dataInput[j,x])
        #Mahalanobis distance metric
        generalMahalanobis[j,1] = t((xi - xj)) %*% pMatrix %*% (xi-xj)
        generalMahalanobis[j,2] = dataInput[j,1]
      }
    }
    orderedMahalanobis = generalMahalanobis[order(generalMahalanobis[,1], decreasing = TRUE),]
    #filling the nieghborhoods
    while (neighbour < k*2){
      if (orderedMahalanobis[neighbour, 2] == dataInput[i,1]){
        compactNeighborhood[i] = compactNeighborhood[i] + orderedMahalanobis[neighbour, 1]
        neighbour = neighbour + 1
      }else{
        scatterNeighborhood[i] = scatterNeighborhood[i] + orderedMahalanobis[neighbour, 1]
        neighbour = neighbour + 1
      }
    }
  }

  discriminabilityFunction = sum(compactNeighborhood)/sum(scatterNeighborhood)
  return(discriminabilityFunction)
}
```

Imagen 3. Evaluación de la función objetivo.

Por último se implementaron las heurísticas Simulated Annealing y algoritmos genéticos.

Algoritmos genéticos

Para los algoritmos genéticos se utilizó el paquete “genalg”[2] modificado para soportar las reglas de los cromosomas. Los cromosomas se definieron como la matriz triangular superior de una matriz P convertidas en vectores. Como el problema se trató de manera continua, los cromosomas no se codificaron en binario, ni tampoco se estandarizaron ya que todos los genes tienen valores entre 0 y 1. La probabilidad de recombinación fue de 80% y la probabilidad de mutación fue de 1%, y por último se implementó un elitismo del 20% del tamaño de la población.

[2] "Package 'genalg' - R." 16 mar.. 2015, https://cran.r-project.org/web/packages/genalg/genalg.pdf. Se consultó el 25 abr.. 2017.

Como reglas de parada se estableció un número de iteraciones en 10.

```
# start with an random population
population = matrix(nrow=popSize, ncol=vars);
# fill values
population[,1:vars] = pSample()
```

Imagen 4. Generación de la población con la función pSample().

```
#Execution of genetic algorithms with
#population size = 3500, iterations = 10, mutation probability = 0.01 and elitism = 20% by defect.
results = rbgaCustom(pSample = getPSample, minGen, maxGen, evalFunc = objectiveFunction,
                     popSize = 3500, iters = 10, mutationChance = 0.01, verbose = FALSE)
```

Imagen 5. Ejecución de algoritmos genéticos.

Simulated Annealing

Para la metaheurística Simulated Annealing, se tomaron los siguientes parámetros:

- Temperatura inicial = 1.
- Temperatura final = 0.1.
- Número de iteraciones = 50
- Factor de enfriamiento = 0.99

La muestra inicial se generó con el algoritmo presentado en las imágenes 1 y 2. La generación del vecino se realizó cambiando las variables con una distribución normal con media en la variable anterior. El algoritmo utilizado es el siguiente:

```
simulated_annealing <- function(pSample, objectiveFunction, tempIni = 1, tempEnd = .1, niter = 50, coolingFactor = .99) {
  bestSolution <- currentSolution <- neighSolution <- s0 <- pSample()
  bestOutput <- currentOutput <- neighOutput <- objectiveFunction(s0)

  for (k in 1:niter) {
    temp = tempIni
    message("It\tBest\tCurrent\tNeigh\tTemp")
    message(sprintf("%i\t%.4f\t%.4f\t%.4f\t%.4f", 0L, bestOutput, currentOutput, neighOutput, tempIni))


    while (temp > tempEnd) {
      temp <- (temp * coolingFactor)
      # consider a random neighbor
      neighSolution <- pSample(currentSolution)
      neighOutput <- objectiveFunction(neighSolution)
      # update current state
      if (neighOutput < currentOutput || runif(1, 0, 1) < exp(-(neighOutput - currentOutput) / temp)) {
        currentOutput <- neighOutput
        currentSolution <- currentSolution
        # update best state
        if (neighOutput < bestOutput) {
          bestSolution <- neighSolution
          bestOutput <- neighOutput
        }
      }
      message(sprintf("%i\t%.4f\t%.4f\t%.4f\t%.4f", k, bestOutput, currentOutput, neighOutput, temp))

    }
  }

  return(list(iterations = niter, best_value = bestOutput, best_state = bestSolution))
}
```

Imagen 6. Algoritmo de Simulated Annealing.

Resultados

El tiempo de ejecución de ambas heurísticas se calculó con la función de R llamada system.time.

Algoritmos Genéticos

Se hicieron dos ejecuciones de algoritmos genéticos con diferentes parámetros y en ambas se evaluó el tiempo de ejecución y la calidad de la solución.

En la primera ejecución se establecieron como parámetros una población de 6000 que es casi el doble del tamaño de los cromosomas, cinco iteraciones, probabilidad de mutación del 1% y elitismo del 20%.
La ejecución tardó 2.55 horas y la mejor solución fue de 1.0123.

```
> source('~/LSMLH/heuristica1.R')
bestOutput:  1.012515 Mean:  1.018512
bestOutput:  1.012455 Mean:  1.017639
bestOutput:  1.01245 Mean:  1.01687
bestOutput:  1.012297 Mean:  1.016208
bestOutput:  1.012297 Mean:  1.015647
```

Imagen 7. Mejor solución y media en cada iteración de la primera ejecución de GA.

En la segunda ejecución se disminuyó la población a 3500 y se aumentaron las iteraciones a 10. El resto de parámetros se dejaron igual.
La ejecución se tardó 2.61 horas y la mejor solución fue 1.02389.

```
> source('~/Documentos/Metaheuristics/LocallySupervisedMetricLearningHeuristics/heuristica1.R')
bestOutput:  1.024015 Mean:  1.03162
bestOutput:  1.024015 Mean:  1.030676
bestOutput:  1.024002 Mean:  1.029637
bestOutput:  1.023969 Mean:  1.028687
bestOutput:  1.023905 Mean:  1.028024
bestOutput:  1.023905 Mean:  1.027392
bestOutput:  1.023905 Mean:  1.026832
bestOutput:  1.023902 Mean:  1.026397
bestOutput:  1.023894 Mean:  1.025977
bestOutput:  1.023887 Mean:  1.02566
```

Imagen 8. Mejor solución y media en cada iteración de la segunda ejecución de GA.

Simulated Annealing

Con Simulated Annealing también se realizaron dos ejecuciones con diferentes parámetros y se evaluó tanto el tiempo de ejecución como la calidad de la solución. En las dos ejecuciones se estableció la temperatura inicial en 1 y la temperatura final en 0.1.

La primera ejecución se realizó con cinco iteraciones y con un factor de enfriamiento de 0.99.
La ejecución se tardó 6.17 minutos y la mejor solución fue 0.9884

5	0.9884	1.0395	1.0395	0.1141
5	0.9884	1.0326	1.0326	0.1129
5	0.9884	1.0346	1.0346	0.1118
5	0.9884	1.0177	1.0177	0.1107
5	0.9884	1.0163	1.0163	0.1096
5	0.9884	1.0338	1.0338	0.1085
5	0.9884	1.0306	1.0306	0.1074
5	0.9884	0.9965	0.9965	0.1063
5	0.9884	1.0326	1.0326	0.1053
5	0.9884	1.0151	1.0151	0.1042
5	0.9884	1.0318	1.0318	0.1032
5	0.9884	1.0343	1.0343	0.1021
5	0.9884	1.0247	1.0247	0.1011
5	0.9884	1.0287	1.0287	0.1001
5	0.9884	1.0372	1.0372	0.0991

Imagen 9. Resultados primera ejecución de SA. La primera columna es el número de iteración, la segunda la mejor solución, la tercera es la solución de la corrida anterior, la cuarta indica la solución en la corrida actual y la última es la temperatura.

En la segunda ejecución se establecieron como parámetros el número de iteraciones en 10 y el factor de enfriamiento en 0.9.
La ejecución se tardó 1,20 minutos y la mejor solución fue 0.9983

10	0.9983	1.0668	1.0668	0.4783
10	0.9983	1.0325	1.0325	0.4305
10	0.9983	1.0330	1.0330	0.3874
10	0.9983	1.0358	1.0358	0.3487
10	0.9983	1.0303	1.0303	0.3138
10	0.9983	1.0391	1.0391	0.2824
10	0.9983	1.0332	1.0332	0.2542
10	0.9983	1.0376	1.0376	0.2288
10	0.9983	1.0366	1.0366	0.2059
10	0.9983	1.0321	1.0321	0.1853
10	0.9983	1.0118	1.0118	0.1668
10	0.9983	1.0289	1.0289	0.1501
10	0.9983	1.0141	1.0141	0.1351
10	0.9983	1.0376	1.0376	0.1216
10	0.9983	1.0378	1.0378	0.1094
10	0.9983	1.0352	1.0352	0.0985

Imagen 10. Resultados segunda ejecución de SA. La primera columna es el número de iteración, la segunda la mejor solución, la tercera es la solución de la corrida anterior, la cuarta indica la solución en la corrida actual y la última es la temperatura.

Conclusiones

Para este problema específico fue mucho más eficiente Simulated Annealing (SA) que los algoritmos genéticos (GA). SA logró obtener una mejor solución en un tiempo mucho menor en las dos ejecuciones realizadas. En cuanto a GA, se puede apreciar que variar el tamaño de la población no afectó demasiado la tasa de disminución en la función objetivo. Mientras que en SA se pudo notar que el parámetro más sensible es el factor de enfriamiento, aún más que el número de iteraciones.