Exercício 7: Problema de Classificação

Rúbia Reis Guerra
2013031143

14 de junho de 2017

1 Introdução

Nesta atividade, foi proposta a escolha e solução de um problema resolvível por redes neurais artificiais. Para tanto, deve-se escolher um *dataset* representativo do problema e três métodos diferentes de treinamento. Ao final, deve-se comparar as soluções utilizando uma métrica apropriada para o tipo de problema.

2 Descrição do Problema

A base de dados escolhida foi o dataset *DNA*, do pacote *mlbench*. A seguir pode-se observar o resumo da base:

```
> rm(list=ls())
> library("RSNNS")
> library("mlbench")
> data(DNA)
> # Quantidade de exemplos e atributos #
> dim(DNA[,1:180])

[1] 3186  180

> # Quantidade e resumo dos rótulos #
> summary(DNA$Class)

  ei   ie    n
 767  765 1654
```

Para esta base, será resolvido um problema de classificação, cujo objetivo é encontrar a classe a qual um novo exemplo pertence, dentre as categorias já existentes (ei,ie,n).

3 Descrição da Solução

A solução proposta consiste em aleatorizar os índices das observações, dividir o dataset em conjuntos de treino (70% dos dados) e teste (30% dos dados) e realizar o treinamento utilizando métodos do pacote *RSNNS*, descritos nas próximas seções. Serão realizados 10 treinamentos para cada método e, ao final, será feita uma comparação dos resultados de classificação para cada técnica utilizada.

3.1 Treinamento por BackpropWeightDecay

O algoritmo de backpropagation faz ajustes calculando a derivada ou a inclinação do erro da rede em relação à saída de cada neurônio. Ele tenta minimizar o erro geral ao diminuir essa inclinação para o valor mínimo para cada peso, avançando um passo abaixo da inclinação de cada época. Se a rede tomar medidas que muito grandes, pode passar o mínimo global. Se tomar medidas muito pequenas, pode estabelecer-se em mínimos locais, ou gastar uma quantidade excessiva de tempo para chegar ao mínimo global. O Weight Decay foi introduzido por P. Werbos, em 1988. A ideia do algoritmo é simples: em cada iteração do ciclo de treinamento, após a atualização dos valores de pesos e bias da rede neural, os pesos e os biases são diminuídos por uma pequena quantidade. Isso tende a manter as amplitudes do peso e dos valores de bias pequenos, o que, por sua vez, impede o overfitting.

3.2 Treinamento por Resilient Backpropagation (RProp)

Rprop, ou resilient backpropagation, é uma heurística para aprendizagem supervisionada em redes neurais artificiais feedforward, criada por Martin Riedmiller e Heinrich Braun em 1992. A heurística leva em consideração apenas o sinal da derivada parcial sobre todos os padrões (e não a magnitude), e age de forma independente em cada peso. Para cada peso, se houver uma mudança de sinal da derivada parcial da função de erro total em comparação com a última iteração, o valor de atualização para esse peso é multiplicado por um fator η^-, onde $\eta^- < 1$. Se a última iteração produzir mesmo sinal, o valor de atualização é multiplicado por um fator de η^+, onde $\eta^+ > 1$. Assim, são calculados os valores de atualização para cada peso, e, por fim, cada peso é alterado pelo seu próprio valor de atualização, na direção contrária da derivada parcial do peso, de modo a minimizar a função de erro total. η^+ é configurado empiricamente para 1.2 e η^- para 0.5. O algoritmo Rprop possui três parâmetros: o valor de atualização inicial, um limite para o tamanho máximo do passo e o expoente de decaimento de peso.

3.3 Treinamento por SCG

O algoritmo básico de backpropagation ajusta os pesos na direção de descida mais íngreme (contrária a do gradiente). Esta é a direção em que a função de

custo está diminuindo com mais rapidez. Embora a função diminua mais rapidamente ao longo do negativo do gradiente, isso não produz necessariamente a convergência mais rápida. Nos algoritmos de gradiente conjugado, uma busca é realizada ao longo das direções conjugadas, o que produz convergência geralmente mais rápida do que as direções de descida mais íngremes. Na maioria dos algoritmos de gradiente conjugado, o tamanho do passo é ajustado em cada iteração. Uma busca é feita ao longo da direção do gradiente conjugado para determinar o tamanho da etapa, que minimiza a função de desempenho ao longo dessa linha. Porém, a busca por essa linha é computacionalmente dispendiosa, uma vez que exige que a resposta da rede a todas as entradas de treinamento seja calculada várias vezes para cada pesquisa. O algoritmo de *Scaled Conjugate Gradient* (SCG), desenvolvido por Moller [Moll93], foi projetado para contornar o problema e tem como ideia básica combinar a abordagem da região modelo-confiança (usada no algoritmo Levenberg-Marquardt), com a abordagem do gradiente conjugado.

4 Implementação

```
> rm(list=ls())
> # Bibliotecas utilizadas #
> library("RSNNS") # Função de treinamento
> library("mlbench") # Dataset
> library(pROC) # Metrica
> data(DNA) # Dataset DNA
> nIter <- 10 # Número de iterações
> aucrprop <- c()
> aucscg <- c()
> aucbpwd <- c()
> for(i in 1:nIter){
+   # Amostragem de dados #
+   dna <- DNA[sample(1:nrow(DNA),length(1:nrow(DNA))),1:ncol(DNA)]
+   dna[is.na(DNA)] <- 0
+   dnaValues <- data.matrix(DNA[,1:180])
+   dnaTargets <- decodeClassLabels(as.numeric(DNA$Class))
+
+   # Divisão dos conjuntos de treino e teste #
+   dna <- splitForTrainingAndTest(dnaValues, dnaTargets, ratio=0.30)
+   dna <- normTrainingAndTestSet(dna)
+
+   # Treinamento
+   # RProp #
+   rprop<-mlp(dna$inputsTrain, dna$targetsTrain, size=2, maxit=40,
+              inputsTest=dna$inputsTest, targetsTest=dna$targetsTest,
+              initFunc="Randomize_Weights",
+              learnFunc="Rprop",
```

```
+             updateFunc="Topological_Order",
+             updateFuncParams=c(0),
+             hiddenActFunc="Act_Logistic",
+             shufflePatterns=TRUE, linOut=FALSE)
+   # SCG #
+   scg<-mlp(dna$inputsTrain, dna$targetsTrain, size=2, maxit=40,
+             inputsTest=dna$inputsTest, targetsTest=dna$targetsTest,
+             initFunc="Randomize_Weights",
+             learnFunc="SCG",
+             updateFunc="Topological_Order",
+             updateFuncParams=c(0),
+             hiddenActFunc="Act_Logistic",
+             shufflePatterns=TRUE, linOut=FALSE)
+
+   # Backpropagation com Weight Decay #
+   bpwd<-mlp(dna$inputsTrain, dna$targetsTrain, size=2, maxit=40,
+             inputsTest=dna$inputsTest, targetsTest=dna$targetsTest,
+             initFunc="Randomize_Weights",
+             learnFunc="BackpropWeightDecay",
+             updateFunc="Topological_Order",
+             updateFuncParams=c(0),
+             hiddenActFunc="Act_Logistic",
+             shufflePatterns=TRUE, linOut=FALSE)
+   # Teste #
+   yrprop <- predict(rprop,dna$inputsTest)
+   yscg <- predict(scg,dna$inputsTest)
+   ybpwd <- predict(bpwd,dna$inputsTest)
+
+   # ROC #
+   rocrprop <- multiclass.roc(dna$targetsTest, yrprop)
+   rocscg <- multiclass.roc(dna$targetsTest, yscg)
+   rocbpwd <- multiclass.roc(dna$targetsTest, ybpwd)
+
+   # AUC #
+   aucrprop[i] <- rocrprop$auc
+   aucscg[i] <- rocscg$auc
+   aucbpwd[i] <- rocbpwd$auc
+ }
```

5 Resultados

Para comparar o desempenho dos classificadores, será usada a área abaixo da curva ROC (AUC) para múltiplas classes, implementada no pacote *pROC*. De forma resumida, a área sob a curva ROC especifica a probabilidade de que, quando é escolhido um exemplo positivo e um negativo ao acaso, a função de

decisão atribui um valor maior ao positivo do que ao exemplo negativo. Assim, tem-se um "melhor"resultado quando a área aproxima-se de 1:

```
> results <- cbind(c(mean(aucrprop),mean(aucscg),mean(aucbpwd))*100,c(sd(aucrprop),sd(aucscg
> results <- round(results,2)
> colnames(results) <- c('Média (%)','Std. (%)')
> rownames(results) <- c('RProp','SCG','Backpropagation w/ W.D.')
> results

                        Média (%) Std. (%)
RProp                       95.58     1.70
SCG                         97.08     0.71
Backpropagation w/ W.D.     96.90     0.49
```