Modelos baseados em instâncias

Guia Prático

Luís Torgo
FEP, Universidade do Porto
ltorgo@liacc.up.pt

18 de Novembro de 2004

Este guia tem como objectivo ajudar os(as) alunos(as) na utilização do R para obter modelos baseados em instâncias.

1. Problemas de Classificação

Começemos por analisar como obter modelos baseados em instâncias para problemas de classificação, ou seja problemas em que pretendemos classificar os casos de teste numa de x possíveis classes.

A função `knn` da package *class* permite-nos obter este tipo de modelos.

Vejamos então um exemplo de utilização. Começemos por carregar o package:

```
> library(class)
```

Na nossa exemplificação deste tipo de modelos, vamos usar os conhecidos dados *iris*. Começemos por os carregar e ver algumas das suas estatísticas descritivas básicas,

```
> data(iris)
> summary(iris)
```

```
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
 Median :5.800   Median :3.000   Median :4.350   Median :1.300
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
       Species
 setosa    :50
 versicolor:50
 virginica :50
```

Se quiser saber mais alguma informação sobre este conjunto de dados poderá obter ajuda sobre eles com o seguinte comando,

```
> help("iris")
```

Vamos agora obter uma divisão aleatória dos 150 casos nos dados *iris*. Esta operação tem como objectivo obter dois conjuntos de dados separados, um para obter o modelo, e outro para o testar. Esta uma das possíveis formas de obter estimativas mais fiáveis sobre a real performance de um qualquer modelo. Vamos usar 70% dos casos para obter o modelo e os restantes 30% para o testar. O código seguinte cria dois data frames com estas características,

```
> train.idxs <- sample(nrow(iris), as.integer(0.7 * nrow(iris)))
> train <- iris[train.idxs, ]
> test <- iris[-train.idxs, ]
```

A primeira linha obteve um vector de 105 números que são as 70% linhas do data frame *iris* que foram escolhidas aleatoriamente (através da função `sample`) para pertencerem ao conjunto de dados de treino (isto é os dados usados para obter o modelo). As duas seguintes instruções criam exactamente os dois data frames mencionados anteriormente usando o vector aleatório de linhas obtido pela função `sample`. No caso de não entender completamente o funcionamento destas linahs de código procure executá-las passo a passo para melhor as compreender.

Vamos agora obter uma divisão aleatória dos 150 casos nos dados *iris*. Esta operação tem como objectivo obter dois conjuntos de dados separados, um para obter o modelo, e outro para o testar. Esta uma das possíveis formas de obter estimativas mais fiáveis sobre a real performance de um qualquer modelo. Vamos usar 70% dos casos para obter o modelo e os restantes 30% para o testar. O código seguinte cria dois data frames com estas características,

```
> train.idxs <- sample(nrow(iris), as.integer(0.7 * nrow(iris)))
> train <- iris[train.idxs, ]
> test <- iris[-train.idxs, ]
```

A primeira linha obteve um vector de 105 números que são as 70% linhas do data frame *iris* que foram escolhidas aleatoriamente (através da função `sample`) para pertencerem ao conjunto de dados de treino (isto é os dados usados para obter o modelo). As duas seguintes instruções criam exactamente os dois data frames mencionados anteriormente usando o vector aleatório de linhas obtido pela função `sample`. No caso de não entender completamente o funcionamento destas linahs de código procure executá-las passo a passo para melhor as compreender.

Vamos agora passar aos modelos baseados em instâncias. Como vimos nas aulas téoricas este tipo de modelos não obtêm de facto qualquer modelo. Assim a utilização destes modelos vai passar unicamente por obter previsões. Vejamos então como proceder, usando as amostras obtidas acima,

```
> prevs.1NN <- knn(train[, -5], test[, -5], train[, 5])
```

A função `knn` tem vários parâmetros (como poderá observar fazendo `help('knn')`). No exemplo acima usamos os seus valores por defeito, nomeadamente quanto ao número de vizinhos a usar para obter as previsões, cujo valor por defeito é 1 vizinho. Infelizmente, esta função sai um pouco do habitual nas funções de modelação no R, não permitindo o uso de fórmulas para indicar a forma funcional do modelo. Assim, no primeiro parâmetro, indicamos quais os dados para treino, no segundo parâmetro quais os dados de teste, e no terceiro parâmetro os valores da classe na amostra de treino.

Se quisermos obter uma matriz de confusão das previsões obtidas por este modelo 1-NN, podemos usar,

```
> (conf.mtrx.1NN <- table(prevs.1NN, test[, 5]))

prevs.1NN    setosa versicolor virginica
  setosa     18      0          0
  versicolor  0     12          2
  virginica   0      2         11
```

Note como englobamos a instrução de atribuição por uns parêntesis. Isto permite, além de realizar a atribuição, imediatamente imprimir o seu resultado, sem termos que, numa segunda instrução, perguntar ao R qual o valor do objecto `conf.mtrx.1NN`.

Como podemos observar o modelo 1-NN comporta-se bastante bem, só errando em 4 dos 45, o que leva a uma percentagem de acerto que pode ser calculada usando,

```
> (acc.1NN <- sum(diag(conf.mtrx.1NN))/sum(conf.mtrx.1NN) * 100)

[1] 91.11111
```

O modelo 1-NN acerta portanto em 91.11% dos 45 casos de teste.

Vejamos agora os resultados que obteriamos usando outros valores para o número de vizinhos mais próximos. Tentemos 3 e 5 vizinhos,

```
> prevs.3NN <- knn(train[, -5], test[, -5], train[, 5], k = 3)
> (conf.mtrx.3NN <- table(prevs.3NN, test[, 5]))

prevs.3NN    setosa versicolor virginica
  setosa     18      0          0
  versicolor  0     12          2
  virginica   0      2         11

> (acc.3NN <- sum(diag(conf.mtrx.3NN))/sum(conf.mtrx.3NN) * 100)

[1] 91.11111

> prevs.5NN <- knn(train[, -5], test[, -5], train[, 5], k = 5)
> conf.mtrx.5NN <- table(prevs.3NN, test[, 5])
> (acc.5NN <- sum(diag(conf.mtrx.5NN))/sum(conf.mtrx.5NN) * 100)

[1] 91.11111
```

Vejamos agora os resultados que obteriamos usando outros valores para o número de vizinhos mais próximos. Tentemos 3 e 5 vizinhos,

```
> prevs.3NN <- knn(train[, -5], test[, -5], train[, 5], k = 3)
> (conf.mtrx.3NN <- table(prevs.3NN, test[, 5]))

prevs.3NN    setosa versicolor virginica
  setosa     18      0          0
  versicolor  0     12          2
  virginica   0      2         11

> (acc.3NN <- sum(diag(conf.mtrx.3NN))/sum(conf.mtrx.3NN) * 100)

[1] 91.11111

> prevs.5NN <- knn(train[, -5], test[, -5], train[, 5], k = 5)
> conf.mtrx.5NN <- table(prevs.3NN, test[, 5])
> (acc.5NN <- sum(diag(conf.mtrx.5NN))/sum(conf.mtrx.5NN) * 100)

[1] 91.11111
```

Podemos agora coleccionar os resultados dos 3 modelos num só vector,

```
> resultados <- c(acc.1NN, acc.3NN, acc.5NN)
> names(resultados) <- c("1-NN", "3-NN", "5-NN")
> resultados

    1-NN     3-NN     5-NN
91.11111 91.11111 91.11111

> round(resultados, 3)

  1-NN   3-NN   5-NN
91.111 91.111 91.111
```

2. Problemas de Regressão

Vamos agora ver como obter modelos baseados em instâncias para problemas de regressão. Embora existam várias funções no R que implementam várias variantes destes modelos, vamos usar a função `loess()` pela sua flexibilidade que permite obter vários tipos de modelos através de alguns dos seus parâmetros. Todavia, convém notar que esta função está limitada ao uso de no máximo 4 variáveis predictivas.

Vamos usar os dados `cars` nas nossas experiências práticas. Mais uma vez poderá usar o respectivo *help* para saber mais detalhes sobre estes dados.

2. Problemas de Regressão

Vamos agora ver como obter modelos baseados em instâncias para problemas de regressão. Embora existam várias funções no R que implementam várias variantes destes modelos, vamos usar a função `loess()` pela sua flexibilidade que permite obter vários tipos de modelos através de alguns dos seus parâmetros. Todavia, convém notar que esta função está limitada ao uso de no máximo 4 variáveis predictivas.

Vamos usar os dados `cars` nas nossas experiências práticas. Mais uma vez poderá usar o respectivo *help* para saber mais detalhes sobre estes dados.

Vamos em primeiro lugar, ver como obter diversas variantes da função `loess`, para obter diferentes tipos de modelos baseados em instâncias.

Vamos então obter 3 modelos diferentes: um modelo *kernel* (um polinómio de grau 0); um modelo linear local (um polinómio de grau 1); e um polinómio local de segundo grau.

Começemos pelo modelo *kernel*,

```
> ll0 <- loess(dist ~ speed, data = cars, degree = 0, span = 0.2)
```

O parâmetro `span` permite-nos indicar a largura de banda a usar nas aproximações locais, neste caso 20% da amostra toda. O parâmetro `degree` permite obviamente indicar o grau do polinómio local a usar, neste caso grau 0.

Vejamos agora os outros dois polinómios,

```
> ll1 <- loess(dist ~ speed, data = cars, degree = 1)
> ll2 <- loess(dist ~ speed, data = cars)
```

Neste caso usamos alguns valores por defeito de alguns parâmetros (veja o help da função para saber quais são esses valores).

Vamos agora tentar fazer uma comparação gráfica (Figura 1) do tipo de aproximações dadas por estes modelos, tirando partido do facto de ser um problema só com duas variáveis o que facilita a visualização. Vamos também incluir a aproximação de um modelo linear clássico (global) para contrastar com estas aproximações locais. Atente no uso da função `fitted` para obter os valores previstos pelos modelos para os casos da amostra de treino.

```
> plot(cars$speed, cars$dist, main = "Distância de travagem", xlab = "Velocidade",
+     ylab = "Distância")
> abline(lm(dist ~ speed, data = cars))
> lines(cars$speed, fitted(ll0), col = "red")
> lines(cars$speed, fitted(ll1), col = "blue")
> lines(cars$speed, fitted(ll2), col = "green")
> legend(5, 120, c("lin.regr", "kernel (20%)", "local linear",
+     "local pol. (2)"), col = c("black", "red", "blue", "green"),
+     lty = 1)
```

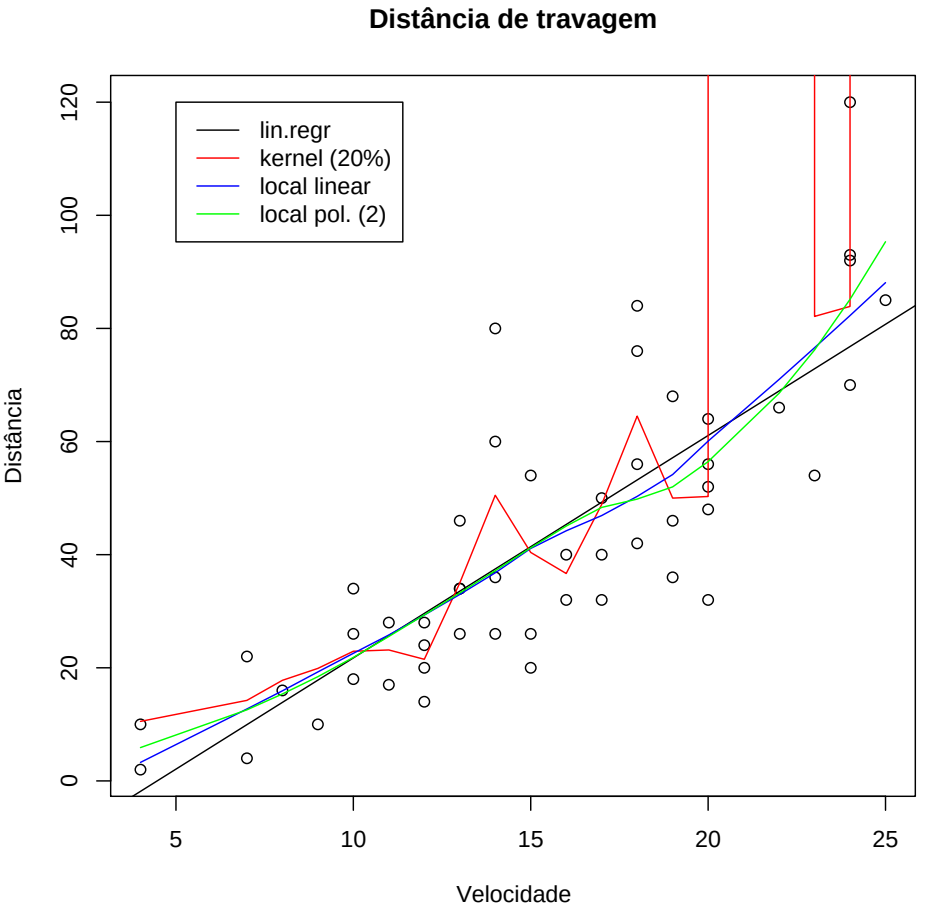

Figura 1: Valores previstos pelos modelos `loc0`, `loc1` e `loc2`.

Em seguida vamos fazer uma comparação semelhante às realizadas anteriormente, sobre uma amostra de teste separada. Começemos por criar as amostras,

```
> train.idxs <- sample(nrow(cars), as.integer(0.7 * nrow(cars)))
> train <- cars[train.idxs, ]
> test <- cars[-train.idxs, ]
```

Em seguida vamos fazer uma comparação semelhante às realizadas anteriormente, sobre uma amostra de teste separada. Começemos por criar as amostras,

```
> train.idxs <- sample(nrow(cars), as.integer(0.7 * nrow(cars)))
> train <- cars[train.idxs, ]
> test <- cars[-train.idxs, ]
```

Vamos agora obter os modelos e suas previsões,

```
> loc0 <- loess(dist ~ speed, train, degree = 0, span = 0.2)
> preds.loc0 <- predict(loc0, test)
> loc1 <- loess(dist ~ speed, train, degree = 1)
> preds.loc1 <- predict(loc1, test)
> loc2 <- loess(dist ~ speed, train, degree = 2)
> preds.loc2 <- predict(loc2, test)
```

Em seguida vamos fazer uma comparação semelhante às realizadas anteriormente, sobre uma amostra de teste separada. Começemos por criar as amostras,

```
> train.idxs <- sample(nrow(cars), as.integer(0.7 * nrow(cars)))
> train <- cars[train.idxs, ]
> test <- cars[-train.idxs, ]
```

Vamos agora obter os modelos e suas previsões,

```
> loc0 <- loess(dist ~ speed, train, degree = 0, span = 0.2)
> preds.loc0 <- predict(loc0, test)
> loc1 <- loess(dist ~ speed, train, degree = 1)
> preds.loc1 <- predict(loc1, test)
> loc2 <- loess(dist ~ speed, train, degree = 2)
> preds.loc2 <- predict(loc2, test)
```

Podemos calcular o erro quadrado médio destas previsões usando,

```
> (mse.loc0 <- mean((preds.loc0 - test[, ncol(test)])^2))

[1] Inf

> (mse.loc1 <- mean((preds.loc1 - test[, ncol(test)])^2))

[1] 450.6404

> (mse.loc2 <- mean((preds.loc2 - test[, ncol(test)])^2))

[1] 520.4206
```

Podemos também ficar com uma ideia do comportamento dos modelos através de uma visualização das previsões. Por exemplo, a Figura 2 mostra-nos os valores previstos e os respectivos valores verdadeiros, relativos aos dados de teste. Esta figura foi obtida com o código,

```
> plot(preds.loc1, test[, ncol(test)], xlab = "Valores previstos",
+     ylab = "Valores previstos", main = "As previsões dos modelos loc0, loc1 e loc2",
+     col = "red", ylim = range(c(preds.loc1, preds.loc2), na.rm = T))
> points(preds.loc0, test[, ncol(test)], col = "blue")
> points(preds.loc2, test[, ncol(test)], col = "green")
> abline(0, 1, lty = 2)
> legend(10, 60, c("loc0", "loc1", "loc2"), col = c("blue", "red",
+     "green"), lty = c(1, 1, 1))
```

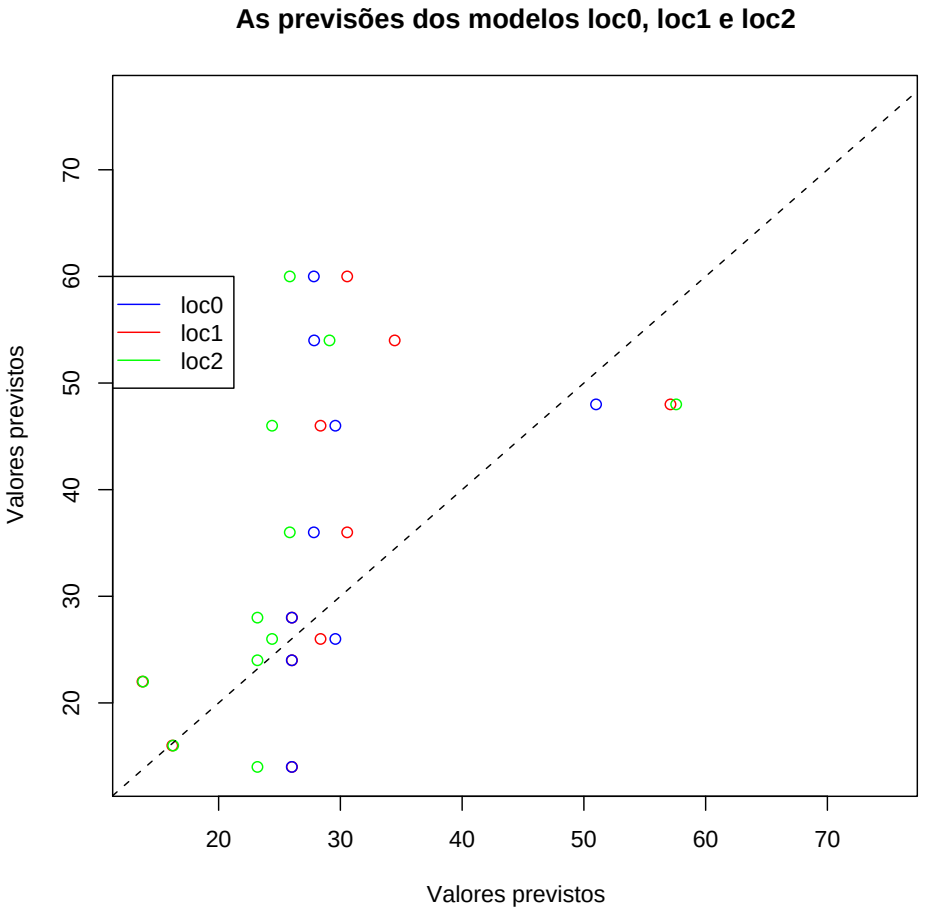

Figura 2: Valores previstos pelos modelos `loc0`, `loc1` e `loc2`.

Se atentarmos nas previsões do modelo `loc2` veremos que existem dois valores `NA`. Para evitar este problema temos que permitir a extrapolação em relação aos valores da amostra de treino. Isso consegue-se obtendo o modelo da seguinte forma,

```
> loc2.extr <- loess(dist ~ speed, train, degree = 2, control = loess.control(surface = "di
> preds.loc2.extr <- predict(loc2.extr, test)
```

3. Exercícios

1. Obtenha uma estimativa mais fiável do erro absoluto médio dos polinómios locais de primeiro e segundo grau sobre o problema `cars`. Para isso construa uma tabela com o erro absoluto médio, valor máximo, valor mínimo e desvio padrão desta estatística em 100 repetições de uma experiência treino+teste resultantes de divisões aleatórias em 70%-30% dos dados originais.

2. Construa um gráfico para mostrar a informação obtida na alínea anterior, de forma visual.