MDLE - Aula Prática 3

Pedro Diogo
Instituto Superior de Engenharia de Lisboa
Mestrado em Engenharia Informática e de Computadores
Lisboa, Portugal
A47573@alunos.isel.pt

Gonçalo Fonseca
Instituto Superior de Engenharia de Lisboa
Mestrado em Engenharia Informática e Multimédia
Lisboa, Portugal
A50185@alunos.isel.pt

I. Visualizar os dados

A.

Ao utilizar a função **sdf_schema** do Spark, esta retorna um dicionário como apresentado na Figura 1.O dicionário mostra cada atributo existente como chave, que inclui dentro de cada um o seu nome e o tipo de variável.

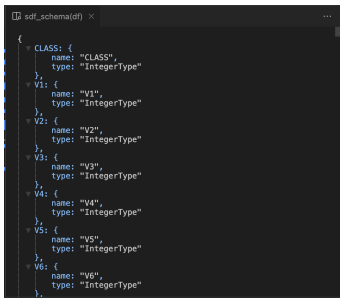

Figura 1: sdf_schema()

B.

Ao chamar a função **head**, esta apresenta as primeiras n linhas do *dataframe* ”df“. Na Figura 2 está apresentado o *output*:

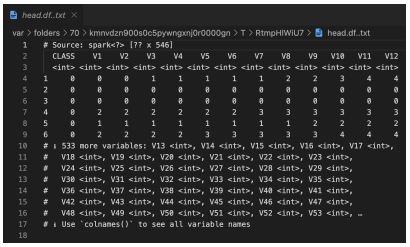

Figura 2: head()

Como podemos ver, a função retorna as 6 primeiras instâncias do dataset e indica o total de colunas (atributos), que neste contexto são 545.

C.

O código disponibilizado que chama a função **stopifnot** encontra-se entre as linhas 50 e 62 do ficheiro **AP3.R** anexado.

O código guarda em variáveis o número de linhas e colunas, usando as funções **sdf_nrow** e **sdf_ncol** do Spark, respetivamente. Depois, guarda em variáveis o número esperado de linhas e de colunas, definido por nós. Por fim, recorrendo à função **stopifnot** são verificadas se as condições entre os valores das variáveis geradas e esperadas são iguais. No caso de existir alguma inconsistência, haverá um *output* com a frase ”XXX == YYY is not TRUE”, onde XXX e YYY são as variáveis criadas.

Posto isto, de forma a garantir que está a funcionar, colocou-se o valor correto no valor expectável das linhas (2910) e de colunas (546, 545 atributos mais 1 coluna para a classe). Como esperado a função validou a dimensão dos dados e prosseguiu com o *script*.

II. Seleção de atributos

A.

O código da linha 69 do ficheiro **AP3.R** serve para reduzir a dataframe *df* para as *features* nos índices da variável idx: 1, 2, 5, 6, 9, 10, 11, 14, 16, 17, 19, 21, 24, 25, 26, 31, 32, 33, 34, 35, 41, 44, 49, 50, 54.

B.

O resultado da seleção do ponto anterior mostra que a dataframe *df.sel* tem as colunas selecionadas de acordo com a variável idx. Pode parecer incorreto mas tendo em conta que o índice 1 corresponde à coluna ”CLASS“, o 2 ao atributo 1, etc, o resultado é o desejado. Na Figura 3 encontra-se o retorno da função **head** na dataframe *df.sel*.

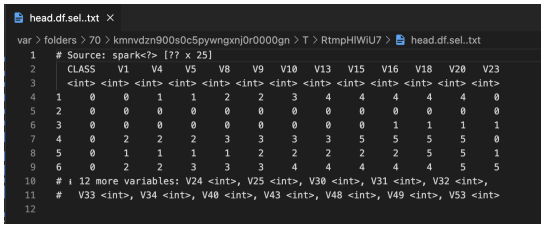

Figura 3: Dataframe df.sel

III. Uso de técnicas genéricas de amostragem

A.

Numa primeira abordagem, definimos a semente (123) para que os resultados fossem consistentes e comparáveis entre as várias técnicas que aplicámos ao dados. Posteriormente definiu-se o peso da divisão de dados de treino (2/3) e teste (1/3) e recorreu-se à função **sdf_random_split** para repartir os dados.

B.

De forma a obter o número de instâncias para cada classe, experimentou-se executar a função **table** para os dois datasets (treino e teste) e usando a função print() para apresentar o resultado na linha de comandos. Ambos os resultados deram o *output* "table of extent 0". Tal acontece visto que as variáveis df.train e df.test são **dataframes do Spark**, e a função **table** usa **dataframes de R ou vetores**.

Para ser possível realizar o pedido neste exercício, o código presente entre as linhas 96 e 104 foi criado. Basicamente usando a função **collect** sobre os dataframes do Spark, passámos a ter dataframe do R. Os resultados das variáveis *train_table* e *test_table* encontram-se na Figura 4.

```
Training dataset counts:

   0    1
1375   75


Test dataset counts:

  0   1
700  40
```

Figura 4: Contagem das classes dos datasets com R.

Para garantir que estes dados são os corretos, fez-se a mesma contagem mas usando o Spark com o código presente entre as linhas 110 a 121 no ficheiro AP3.R.

Os resultados encontram-se na Figura 5.

```
Training dataset counts using spark:
# A tibble: 2 × 2
  CLASS     n
  <int> <dbl>
1     1    75
2     0  1375


Test dataset counts using spark:
# A tibble: 2 × 2
  CLASS     n
  <int> <dbl>
1     1    40
2     0   700
```

Figura 5: Contagem das classes dos datasets com Spark.

C.

Para validar a amostragem feita para os dados treino e teste, utilizou-se um modelo de *Random Forest*.

D.

De seguida, e usando o código disponibilizado no módulo "helperfunctions.R", gerou-se a matriz de confusão para a aprendizagem feita pelo modelo anterior. Os resultados estão presentes na Figura 6.

```
Confusion Matrix and Statistics: Random Forest Model - Baseline

      0   1
  0 698   2
  1  33   7
False Positive Rate : 0.825
Accuracy            : 0.953
Kappa               : 0.271
Pos Pred Value      : 0.175
Neg Pred Value      : 0.997
```

Figura 6: Resultados para *random split*.

IV. Uso de técnicas de amostragem de correção de desequilíbrio

A.

Para realizar *undersampling* do conjunto de treino, foi desenvolvido o código presente entre as linhas 145 e 163. O grande objetivo passa por igualar ou aproximar o a representatividade de ambas as classes (0 e 1). Como existe uma enorme discrepância entre a classe 0 que possui 1375 instâncias e a classe 1 que apenas conta com 75 instâncias, queremos reduzir o número de linhas cuja etiqueta é 0.

Ao aplicar esta técnica, obtivemos 75 instâncias para a classe positiva (classe 1) depois de realizar o *undersampling*. Para a classe negativa (classe 0), obteve-se 67 instâncias. Desta forma temos um número semelhante de instâncias para representar as duas classes.

B.

Posteriormente gerou-se a matriz de confusão para a aprendizagem feita pelo *random forest*. Os resultados estão presentes na Figura 7.

```
Confusion Matrix and Statistics: Random Forest Model - Undersample

      0   1
  0 564 136
  1   8  32
False Positive Rate : 0.200
Accuracy            : 0.805
Kappa               : 0.241
Pos Pred Value      : 0.800
Neg Pred Value      : 0.806
```

Figura 7: Resultados para *undersampling*.

C.

Nesta alínea pretende-se chegar ao mesmo resultado da anterior mas com *oversampling*. Quer isto dizer que vamos aumentar o número de instâncias da classe minoritária (classe 1) de forma a igualar o número de instâncias da classe maioritária.

Com a abordagem descrita acima, obtivemos 1375 instâncias para ambas as classes depois de realizar o *oversampling*.

D.

De seguida gerou-se a matriz de confusão para a aprendizagem feita pelo *random forest* com os novos dados de treino. Os resultados estão presentes na Figura 8.

```
Confusion Matrix and Statistics: Random Forest Model - Oversample

      0   1
  0 610  90
  1   8  32
False Positive Rate : 0.200
Accuracy            : 0.868
Kappa               : 0.341
Pos Pred Value      : 0.800
Neg Pred Value      : 0.871
```

Figura 8: Resultados para *oversampling*.

E.

A última técnica de amostragem de correção de desequilíbrio dos dados utilizada foi o *Boderline-SMOTE*. Este algoritmo tem como objetivo criar dados sintéticos, logo pode ser considerada como uma variante de *oversampling*.

A função **BLSMOTE** da *package* **smotefamily** [1], começa por classificar as observações da classe minoritária em 3 grupos: SAFE/DANGER/NOISE. A classificação olha para o número de vizinhos da classe maioritária para determinar em que grupo se enquadra a observação. Apenas observações classificadas como DANGER são usadas para gerar instâncias sintéticas.

A função recebe os dados em formato de dataframe do R, sem as etiquetas. Estas passadas como o segundo parâmetro da função. Para além destes dois argumentos, é passado um K que representa número de vizinhos mais próximos durante o processo de amostragem, um C que se trata do número de vizinhos mais próximos durante o cálculo do processo de nível seguro e por fim é passado um *method* que pode ser "type1" ou "type2". Estes métodos estão descritos em [2].

A figura 9 representa os resultados obtidos para um K=7, C=5 e o *method*=type1.

```
Confusion Matrix and Statistics: Random Forest Model - Borderline SMOTE

      0   1
  0 675  25
  1  22  18
False Positive Rate : 0.550
Accuracy            : 0.936
Kappa               : 0.400
Pos Pred Value      : 0.450
Neg Pred Value      : 0.964
```

Figura 9: Resultados para BLSMOTE.

V. Comparação dos resultados

Olhando para a Tabela I, podemos ver que para os rácio de falsos positivos, tanto as técnicas de *undersampling* e *oversampling* obtiveram o melhor resultado. Respetivamente à acurácia, obteve-se o melhor resultado sem aplicar qualquer técnica de amostragem. O valor da métrica Kappa mais alto foi para o BLSMOTE. Já o valor mais elevado de predições da classe positiva resultou mais uma vez no empate do *undersampling* e *oversampling*. Por fim o valor mais elevado de predições da classe positiva foi obtido sem fazer amostragem dos dados.

Considerando o problema em questão (classificação utilizando um conjunto de dados desequilibrado), a técnica que obteve melhores resultados, foi o *Bordeline SMOTE*.

	F.P. Rate	Accuracy	Kappa	Pos Pred Value	Neg Pred Value
Baseline	0.825	**0.953**	0.271	0.175	**0.997**
Undersampling	**0.200**	0.805	0.241	**0.800**	0.806
Oversampling	**0.200**	0.868	0.341	**0.800**	0.871
Borderline smote k7	0.550	0.936	**0.400**	0.450	0.964

Tabela I: Métricas de avaliação no uso de técnicas de amostragem de correção de desequilíbrio.

Referências

[1] A Collection of Oversampling Techniques for Class Imbalance Problem Based on SMOTE - [Em linha] [Consult. 21 abr. 2023]. Disponível em https://cran.r-project.org/web/packages/smotefamily/smotefamily.pdf

[2] H. Han, W.-Y. Wang, e B.-H. Mao, ≪Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning≫, em Advances in Intelligent Computing, Berlin, Heidelberg, 2005, pp. 878–887. doi: 10.1007/11538059_91.