INSTITUTO UNIVERSITÁRIO DE LISBOA

2020/2021 (2º semestre)

Regressão Logística e Árvore de Classificação com *Apache Spark*

Processamento de Big Data

Prof. Adriano Lopes

Grupo PL1

Catarina Castanheira Nº 92478
João Martins Nº 93259
Joel Paula Nº 93392

Índice

I - Introdução

Para este trabalho foi escolhido um *dataset* da SWITRS (*Statewide Integrated Traffic Records System*) com informação detalhada sobre todas as colisões reportadas ao CHP (*California Highway Patrol*) desde 1 de janeiro de 2001 até meados de outubro de 2020.

Este *datset* contém um ficheiro em formato *sqlite* com 4 tabelas:

- *Case_ids*: contém os ids dos casos e é usada para construir as outras 3 tabelas;
- *Collisions*: contém informação sobre as colisões;
- *Parties*: contém informação sobre os grupos de pessoas envolvidas na colisão;
- *Victims*: contém informação sobre os ferimentos das pessoas envolvidas na colisão.

O tamanho total do *dataset* é de 5.78 GB.

O nosso objectivo consistiu na implementação de modelos que permitissem a previsão da gravidade de cada colisão registada na tabela Collisions (constituída por 74 colunas e 9 172 565 registos, num total de 3.2 GB[1]), socorrendo-nos dos algoritmos de Regressão Logísitica Multinomial e de Árvore de Classificação disponíveis em https://spark.apache.org/docs/latest/ml-classification-regression.html. Para este efeito, definimos então como variável *target* a *collision_severity*. Esta é uma variável categórica, composta por 5 classes: 1 – *Fatal;* 2 – *Injury (Severe);* 3 – *Injury (Other Visible);* 4 – *Injury (Complaint of Pain);* 0 – *Property Damage Only (PDO).*

NOTAS:

Para consultar o *dataset* original: https://www.kaggle.com/alexgude/california-traffic-collision-data-from-switrs.

Para aceder ao conjunto de dados específico utilizado para este projecto: https://bit.ly/3fri5oV.

Para aceder ao conjunto de dados mais pequeno já limpo e em formato *parquet*: https://bit.ly/3u6zJm3.

Instruções para instalar o *driver* que permite importar ficheiros *sqlite* para o *Apache Spark*:

1. Fazer *download* da *driver* JDBC para SQL Lite a partir de: https://repo1.maven.org/maven2/org/xerial/sqlite-jdbc/3.34.0/sqlite-jdbc-3.34.0.jar
2. Colocar a *driver* em $SPARK_HOME/jars/

[1] Apesar do enunciado do projecto pressupor a utilização de um *dataset* superior a 4GB, obtivemos autorização por parte do Professor Adriano Lopes para usar este de dimensão inferior.

II - Limpeza de Dados e Análise Exploratória

Limpeza dos Dados (ficheiro Jupyter Notebook: Collisions-DC+EDA)

O primeiro passo passou pela análise da tabela *collisions* e seleção dos dados mais relevantes para o nosso objetivo. Analisando o tipo associado a cada tabela, distinguimos facilmente aquelas que estão no tipo string e as que estão noutros formatos (por exemplo, double, long) que podem ser tratados de forma "numérica". Numa primeira abordagem, tratamos as primeiras como "categóricas" (44 colunas) e as segundas como "numéricas" (30 colunas). Durante este passo encontrámos colunas que não tinham informação interessante relativamente ao nosso objetivo ("case_id", "process_date", "officer_id") ou que continham poucos registos com dados ("secondary_ramp", "primary_ramp", "latitude", "longitude", "reporting_district") e por essas razões acabámos por eliminá-las.

O próximo passo consistiu na transformação de algumas colunas que estão classificadas como numéricas em nominais. As variáveis que considerámos para esta transformação foram: "jurisdiction", "caltrans_district", "state_route", "ramp_intersection", "pcf_violation", "postmile".

De acordo com a documentação oficial dos dados (https://tims.berkeley.edu/help/SWITRS.php), as colunas "pedestrian_collision", "bicycle_collision", "motorcycle_collision", "truck_collision", "not_private_property" e "alcohol_involved" são variáveis binárias, sendo que no *dataset* apresentam os valores "1" ou "null". Adicionalmente a variável "not_private_property" aparenta ter apenas dois casos "null" e 9172563 casos referentes a "1", o que nos leva a considerar que não tem variabilidade suficiente e valor adicional para o nosso objetivo (previsão da variável *target* – "collision_severity"). Após confirmarmos que este caso não se verificava para as restantes variáveis anteriores decidimos eliminar apenas a variável "not_private_property" e substituir os valores "null" das variáveis acima por "0".

De seguida fomos analisar os *missing values* nas colunas numéricas e substituímos os valores "null" das colunas "killed_victims" e "injured_victims" pela moda de cada uma (a moda de ambas as colunas é 0, pelo que os valores "null" foram substituídos por 0).

Relativamente às variáveis binárias com *missing values* "intersection", "state_highway_indicator" e "tow_away", adicionámos uma nova categoria com o valor "2" para os representar.

Por fim, para as restantes colunas de variáveis categóricas substituímos os *missing values* por uma nova categoria – "Not Stated". Também definimos o tipo da variável "postmile" como *string*, enquadrando-a no conjunto das variáveis categóricas.

Concluída a limpeza dos dados procedemos à criação de um ficheiro em formato *parquet* com os novos dados limpos. Este é o *dataset* que serve de referência posterior para o treino inicial dos dois modelos de classificação.

Análise dos Dados (ficheiro Jupyter Notebook: Collisions-DC+EDA)

Uma vez que não é possível realizar uma análise correlacional quando incluímos variáveis categóricas, a estratégia encontrada foi fazer a sua transformação para que o *Spark* as pudesse processar como se de numéricas tratassem. Neste sentido, recorremos ao transformador StringIndexer, o qual permite a codificação de colunas do tipo *string* em colunas de índices, que podem então ser adicionadas ao *dataset* e consideradas para a análise correlacional.

Para visualizar as relações entre as variáveis criámos uma matriz de correlações, seguida de uma listagem com uma ordenação decrescente dos valores de correlação de cada variável com a variável *target* – "collision_severity".

De seguida guardámos numa variável os preditores mais promissores, baseando-nos nas correlações e tendo como critério de seleção o valor de correlação com a *target* maior que 0.10.

Entre o conjunto das variáveis com maior correlação com a target se encontravam algumas com uma elevada correlação entre si. Para estes casos concretos, escolhemos manter aquela variável do par que tivesse a maior correlação absoluta com a target. Ficámos então com as seguintes:

Variável preditora	**Correlação com a *target***
"injured_victims"	0.693092
"other_visible_injury_count"	0.620508
"severe_injury_count"	0.426505
"killed_victims"	0.318472
"pedestrian_collision"	0.247537
"tow_away"	0.235786
"motorcyclist_injured_count"	0.233352
"bicyclist_injured_count"	0.198023
"bicycle_collision"	0.185049
"pedestrian_killed_count"	0.161776
"type_of_collisionIndex"	0.161671
"motor_vehicle_involved_withIndex"	0.137209
"pcf_violation_subsectionIndex"	0.129773

"motorcyclist_killed_count"	0.124564

III - Construção dos Modelos de Classificação

Modelo de Regressão Logística Multinomial (*Multinomial Logistic Regression*) (ficheiro Jupyter Notebook: Collisions-LogR)

O processo de criação dos modelos de previsão começa com a importação do ficheiro *parquet* que contém um sub-conjunto de cerca 100 000 registos do *dataset* original. Além deste passo, importamos também algumas bibliotecas que serão necessárias, destacando as que permitem a vectorização de conjuntos de variáveis (*VectorAssembler*), a criação de variáveis *dummy* a partir das variáveis categóricas, e também a definição do próprio modelo de regressão logística.

As funções *showModelSummary* e *showPerformanceOnTest* têm o propósito de implementar métricas de desempenho aos modelos de classificação criados (e que servem de argumento); a primeira, quando temos um modelo a prever as próprias observações do conjunto de treino; a segunda, já quando o modelo foi aplicado na previsão das observações no conjunto de teste.

Como já mencionado acima, a nossa variável alvo (*target*) é a *collision_severity*, variável categórica com 5 classes possíveis. Com a implementação de qualquer um dos modelos (Regressão Logística Multinomial e Árvore de Classificação) é possível a previsão da classe desta variável em cada observação, com um erro/precisão associados. Contudo, para tal ser conseguido, teremos de proceder à transformação de variáveis. E é aqui que entram os *Feature Transformers*: *StringIndexer* (converte as variáveis categóricas em índices, para poderem então ser processadas pelo *OneHotEncoder*), o *OneHotEncoder* (cria um vetor para cada variável categórica que incluamos como input, identificando a presença/ausência de dada classe numa observação através de 1/0) e o VectorAssembler (transformador que agrega a informação das diversas colunas/variáveis num só vetor, que alimentará então o treino e teste do modelo). Por uma questão de eficácia, recorremos à implementação destes transformadores num *pipeline*.

Com a transformação das variáveis conseguida, passamos então para definição de dois conjuntos de dados: o conjunto de treino (*df_train*) com 75% dos registos (75790) e o conjunto de teste (*df_test*) com os restantes 25% dos registos (25418). Na definição destes conjuntos, gerámos um processo pseudoaleatório através do método *randomSplit*().

O primeiro modelo de regressão logística definido contém como preditores todas as variáveis que compõem o *dataset* pré-peparado na fase anterior. Depois de feito o *fit* com os dados de treino e aplicada a função de análise de desempenho do modelo, obtemos uma precisão (*Accuracy*) de 0.627378. Aplicando então o modelo aos dados de teste, obtemos uma precisão de 0.622432 – negligenciavelmente inferior à de treino. Adotando uma estratégia de refinar as *features* que alimentam o modelo, começamos por filtrar aquelas que têm uma correlação acima de 0.20 para a *target*: temos uma precisão de 0.627312 sobre o conjunto de treino e 0.622393 sobre o conjunto de teste. Finalmente, limitamos os preditores ao conjunto *injured_victims*, *other_visible_injury_count* e *severe_injury_count*, variáveis que apresentam as maiores correlações com a *target* (acima de 0.5). Mas não obtivemos variação significativa nos valores de precisão: 0.627312 para o treino e 0.622393 para o teste.

Uma métrica interessante para perceber a utilidade de um algoritmo de classificação é o Índice de *Huberty*, que compara o desempenho daquele face a uma referência simples: a probabilidade da classe modal da variável que pretendemos prever, indicando um valor para a melhoria obtida em relação à técnica de simplesmente escolher a classe modal como previsão para qualquer registo da amostra. Por uma questão prática, incluímos esta métrica dentro das funções *showModelSummary* e *showPerformanceOnTest.* Analisando o índice, percebemos que o algoritmo de regressão logística apresenta resultados muito fracos, com os seguintes índices de Huberty: 0.04928 (modelo 1); 0.052 (modelo 2 e modelo 3).

Árvore de Classificação (*Decision Tree Classifier*) (ficheiro Jupyter Notebook: Collisions-CTree)

A árvore de classificação é um algoritmo bastante comum na classificação. Comparativamente à regressão logística, existe a vantagem de não ser requisito a criação de variáveis *dummy* a partir das variáveis categóricas que tenhamos no *dataset*. Em relação à regressão logística, mantemos somente o requisito de fazer a transformação destas variáveis em índices numéricos através do *StringIndexer*. Contudo, houve um desafio que se levantou na preparação do algoritmo da árvore: percebemos que existia uma limitação no número de classes que o algoritmo tinha capacidade de processar para cada preditor categórico. Neste sentido, foi tomada a decisão de excluir à partida aquelas que se enquadravam neste cenário – *primary_road, secondary_road, collision_date, postmile, beat_number, collision_time, county_city_location, jurisdiction, state_route, e pcf_violation*. Mantivemos as restantes variáveis preditoras que tínhamos em *df_clean* e definimos um *assembler* para as transformar todas num vetor.

O passo seguinte consistiu na construção de um conjunto de treino e um conjunto de teste, na mesma proporção usada nas experimentações com a regressão logística: 75% dos dados para treino e os restantes 25% para teste.

Da mesma maneira que o tínhamos feito na concepção do modelo anterior, para este caso da árvore de classificação voltámos a implementar um sistema de *pipeline*. Na definição do modelo da árvore, alterámos ligeiramente o parâmetro *minInfoGain* (0.05) no sentido de prevenir o *overfitting*. Correndo o modelo na previsão dos dados de treino (*df_train*), obtivemos um desempenho de 0.9946 para a precisão e de 0.9864 para índice de Huberty. Quando aplicamos o modelo na previsão no conjunto de dados de teste, o seu desempenho foi de 0.9939 para a precisão e de 0.9847 para o Índice de *Huberty*.

Aplicando o método toDebugString() ao modelo da árvore, conseguimos obter uma representação da mesma, com a sua "cadeia de decisão". Desta forma, percebemos que as variáveis preditoras utilizadas pelo modelo foram: *injured_victims, killed_victims, severe_injury_count e other_visible_injury_count*.

IV – Conclusão

Na resolução do problema de classificação (prever a gravidade de uma dada colisão, atendendo às variáveis disponíveis) cruzámo-nos com diversos desafios, entre os quais destacamos a dificuldade em encontrar a forma correcta de conseguir importar dados do tipo *sqlite* para o ambiente do *Apache Spark*, a dificuldade inicial em compreender o funcionamento dos transformadores em cada contexto e a sua parametrização ideal, e por fim o maior desafio de todos: o grande tamanho do *dataset* faz-se sentir no elevado tempo de execução das operações e na (in)capacidade da própria máquina virtual em lidar com elas quando tentamos aplicar os modelos de previsão a conjuntos de teste criados por 40% dos dados do *dataset* original. No nosso caso particular, apesar de ser possível a definição e treino do modelo de previsão com os 40% do *dataset* total, tanto na regressão logística como na árvore de classificação, não conseguimos determinar a precisão de cada um deles na previsão dos dados de teste, uma vez que é sistemática a ocorrência de um erro nesta fase: erro catastrófico de perda de ligação ao *Apache Spark* – este deixa de estar acessível.

Com base nos resultados obtidos do desempenho de cada um dos algoritmos, conclui-se que para o âmbito do problema definido – a previsão multi-classe de uma variável categórica – o algoritmo com maior capacidade para o efeito é o da árvore de classificação (aprox. 0.62 de precisão contra aprox. 0.99, respectivamente).