Mammographic Masses

Estudo do *dataset* utilizando Redes Neuronais Artificiais

Grupo 6

André Pereira (pg38923)
Carlos Lemos (pg38410)
João Barreira (a73831)
Rafael Costa (a61799)

Abril 2019

Universidade do Minho
Escola de Engenharia

Trabalho Prático 1
Sistemas Inteligentes – Computação Natural
MEI / MIEI – Universidade do Minho

1 Introdução

Este trabalho prático incide em diversos aspectos cruciais para o desenvolvimento de modelos computacionais capazes de adquirir conhecimento ao incidir sob uma amostra de dados, fazendo um reconhecimento de padrões, produzindo assim uma solução aproximada para um dado problema que poderá ter um nível de complexidade elevado, que poderia exigir um maior custo computacional caso o problema fosse implementado utilizando técnicas tradicionais de programação.

Assim, para o desenvolvimento deste projeto foram utilizado diversos procedimentos, como a preparação e análise de *datasets*, treino e validação de modelos de aprendizagem, especificamente de Redes Neuronais Artificiais *(ANN)*, e otimização de parâmetros do modelo de aprendizagem.

Para tal, foi realizada a implementação de um modelo preditivo *deep learning*, desenvolvido em *Python*, que faz a deteção de massas malignas no tecido mamário. O algoritmo começa por fazer uma análise de um ensaio clínico real, fazendo uma aprendizagem desse conjunto de dados clínicos de casos reais. Este *dataset* apresenta um conjunto de métricas analisadas durante exames de mastografia, tentando assim determinar se as massas detalhadas são benignas ou malignas, intervindo assim num rastreio de células para determinar se um indivíduo possui cancro mamário.

2 *Mammographic Masses Dataset*

O *Mammographic Masses Dataset* corresponde a um conjunto de dados clínicos resultantes de 961 exames de mastografia/mamografia. Este tipo de exame tem como objetivo analisar o tecido mamário por forma a detetar possíveis nódulos (i.e. massas) que se tenham formado, mesmo que não sejam ainda palpáveis. Desta forma – e a partir de vários fatores –, será possível tentar prever a existência de um eventual cancro (caso a massa seja maligna).

Este *dataset* possui as seguintes métricas:

- **BI-RADS** – Corresponde ao valor da avaliação *Breast Imaging-Reporting and Data System* (incompleto=0, negativo=1, achado benigno=2, provavelmente benigno=3, suspeita de anormalidade=4, altamente sugestivo de malignidade=5, malignidade comprovada através de biópsia=6). Esta métrica não foi contemplado na resolução deste estudo.
- ***Age*** – Corresponde à idade do paciente (valor inteiro).
- ***Shape*** – Corresponde à forma da massa/nódulo encontrada (redonda=1, oval=2, lobular=3, irregular=4).
- ***Margin*** – Corresponde à margem da massa/nódulo encontrada (circunscrito=1, micro-lobulado=2, obscurecido=3, mal-definido=4, espiculado=5).
- ***Density*** – Corresponde à densidade da massa/nódulo encontrada (alto=1, médio=2, baixo=3, contém gordura=4)
- ***Severity*** – Corresponde ao grau de severidade da massa/nódulo encontrada (benigno=0, maligno=1).

3 *Data Acquisition*

Esta fase não foi contemplada durante este trabalho prático visto que nos foi fornecido o *dataset* com o qual teríamos de trabalhar, cuja descrição pormenorizada se encontra na **Secção 2**.

4 *Data Visualization*

Estando apresentado o *dataset* e as respetivas métricas, procedeu-se à leitura do *dataset* (na qual se indicou que as células sem valor se encontravam denotadas por um ponto de interrogação) e exclusão da coluna correspondente à métrica *BI-RADS*, como indica a **Figura 1**.

```
# Read dataset from CSV file

df = pd.read_csv('mammographic_masses.data.txt',
                 names=['BI-RADS', 'Age', 'Shape', 'Margin', 'Density', 'Severity'],
                 na_values=['?'])

# Drop unrequired feature 'BI-RADS'

df = df.drop(['BI-RADS'], axis=1)
```

Figura 1: Leitura do *dataset*

De seguida, procedeu-se à impressão de uma tabela para um ficheiro auxiliar *dataset_description.txt* (**Figura 3**) que contém variadas informações estatísticas acerca da distribuição dos valores de cada uma das métricas do *dataset*, como indica a **Figura 2**.

```
# Print dataset description to file

pd.options.display.max_columns = 2000
print(df.describe(), file=open("dataset_description.txt", 'w'))
```

Figura 2: Geração do ficheiro *dataset_description.txt*

	Age	Shape	Margin	Density	Severity
count	956.000000	930.000000	913.000000	885.000000	961.000000
mean	55.487448	2.721505	2.796276	2.910734	0.463059
std	14.480131	1.242792	1.566546	0.380444	0.498893
min	18.000000	1.000000	1.000000	1.000000	0.000000
25%	45.000000	2.000000	1.000000	3.000000	0.000000
50%	57.000000	3.000000	3.000000	3.000000	0.000000
75%	66.000000	4.000000	4.000000	3.000000	1.000000
max	96.000000	4.000000	5.000000	4.000000	1.000000

Figura 3: Tabela com valores estatísticos acerca das métricas do *dataset*

Além disso, por forma obter uma visualização mais agradável da distribuição dos valores dos dados, foram também gerados os gráficos de distribuição para cada uma das métricas do *dataset* (figuras presentes no **Anexo 10.1**) para a pasta *dist*, como indica a **Figura 4**.

```
# Generate feature distribution graphs to files inside dist folder

mkdir("dist")

nRows = df.shape[0]
nBins = int(round(sqrt(nRows)))  # binning

for key in df.keys():
    df.hist(column=key, bins=nBins)
    fig_name = "dist-" + key + ".png"
    plt.savefig("dist/" + fig_name)
```

Figura 4: Geração dos gráficos de distribuição das métricas do *dataset*

Como podemos constatar pela tabela da **Figura 3** e pelos gráficos de distribuições presentes nas figuras do **Anexo 10.1**, as *features* possuem uma distribuição de valores em intervalos e escalas bastante diferentes, pelo que será necessário normalizar (ou standardizar) os seus valores para um estudo mais eficaz.

5 *Data Preprocessing*

Como pudemos constatar através da análise dos dados presente na **Secção 4**, as *features* possuem uma distribuição de valores em intervalos e escalas bastante diferentes, pelo que será necessário normalizar (ou standardizar) os seus valores.

Procedemos então à standardização dos valores das *features* através do recurso ao método *StandardScaler* da biblioteca *scikit-learn*, por forma a termos os valores com uma média e *unit variance* iguais a 0, como indica a **Figura 5**.

```
# Standardize feature data

scaler = StandardScaler()
features[features.columns] = scaler.fit_transform(features[features.columns])  # scales the existing df
```

Figura 5: Standardização dos valores das *features*

Além disso, foi-nos dito no enunciado para retirarmos as linhas do *dataset* correspondentes aos registos com valores em falta, o que foi uma estratégia indicada pois, após os analisarmos, concluímos que estes aparentavam ter um distribuição aleatória pelo conjunto de dados. A **Figura 6** indica o código referente a este processo.

```
# Discard lines with NaN values

df = df.dropna()
```

Figura 6: Standardização dos valores das *features*

6 Construção do Modelo

Para a construção da Rede Neuronal Artificial correspondente ao modelo que serviu de base ao estudo do *dataset*, foi utilizada a biblioteca *Keras* que foi desenvolvida em cima do *Tensorflow*, oferecendo algumas outras funcionalidades. Mais concretamente, foi utilizado o *wrapper KerasClassifier* que permite tirar partido das funcionalidades da biblioteca *scikit-learn*, a qual já tínhamos tido oportunidade de estudar o semestre passado.

Assim sendo, a escolha da biblioteca *Keras* deveu-se sobretudo ao facto de nos ter tornado a utilização da técnica de *cross-validation* num processo bastante mais simples.

O processo de criação da Rede Neuronal Artificial é também ele bastante simples utilizado o *Keras*. Para isso, basta apenas criar uma instância do *KerasClassifier*, passando-lhe uma função de criação do modelo, como indica a **Figura 7**.

```
# Create Keras classifier model

classifier = KerasClassifier(build_fn=create_model, verbose=0)
```

Figura 7: Criação do *KerasClassifier*

Esta função de criação do modelo recebe como argumento os valores dos hiper-parâmetros a utilizar. Depois cria um modelo do *Keras* do tipo *Sequential*, adicionando uma primeira camada *Dense* de *input* com 4 nós (correspondentes às 4 *features*) e uma camada *Dense* de output com 1 nó (correspondente ao *target*). Para cada uma das camadas intermédias (hiper-parâmetro *hidden_layers*), cria uma camada *Dense* com o número de nós igual ao valor recebido no hiper-parâmetro *nodes_per_layer*. Por fim, é compilado o modelo utilizando um otimizador *Adam* com a taxa de aprendizagem recebida como argumento (hiper-parâmetro *learning_rate*), *loss* segundo *mean_squared_error* (MSE) e métrica de *accuracy*. Para cada uma das camadas criadas é ainda definida a função de ativação como sendo a recebida como argumento (hiper-parâmetro *activation_fn*). Este processo encontra-se demonstrado na **Figura 8**.

```
def create_model(hidden_layers=2, nodes_per_layer=3, activation_fn='relu', learning_rate=1e-2):
    model = Sequential()
    model.add(Dense(4, activation=activation_fn, input_shape=(4,)))  # input layer

    for i in range(hidden_layers):
        model.add(Dense(nodes_per_layer, activation=activation_fn))

    model.add(Dense(1, activation=activation_fn))  # output layer

    adam = Adam(lr=learning_rate)
    model.compile(loss='mean_squared_error', optimizer=adam, metrics=['accuracy'])

    return model
```

Figura 8: Função de criação do modelo

Como foi dito anteriormente, o processo de *cross-validation* torna-se bastante simples pelo facto de se poder tirar partido do método correspondente presente na biblioteca *scikit-learn*. A **Figura 9** apresenta a codificação de uma *cross-validation* do tipo *KFold* (tipo *default*) para k=10 *folds* (no contexto da otimização *Random Search*).

```
random_search = RandomizedSearchCV(classifier, param_distributions=hp_dist, n_iter=20, cv=10)
random_search.fit(features, target)
```

Figura 9: *Cross-validation* do tipo *KFold* com k=10 (no contexto do *Random-Search*)

7 Hiper-parametrização *Random Search* (ficheiro *random_search.py*)

De seguida, começámos por efetuar a primeira tentativa de hiper-parametrização do modelo criado. Devido ao facto de podermos utilizar as funcionalidades da biblioteca *scikit-learn*, decidimos, então, começar por efetuar uma otimização do tipo *Random Search* tirando partido do método *RandomSearchCV*.

Para isso, tivemos de criar um dicionário que possuísse as distribuições dos valores para os hiper-parâmetros a testar:

- ***hidden_layers*** – potências de 2 (escolhemos [2, 4, 8, 16, 32]);
- ***nodes_layer*** – intervalo de inteiros entre 1 a 20;
- ***activation_fn*** – ReLU ou Sigmoid;
- ***learning_rate*** – intervalo entre 1e-8 e 1e-2.

Depois, bastou-nos chamar o método *RandomSearchCV* com o dicionário das distribuições criado, e definir o número de iterações (mantendo um *cross-validation* do tipo *KFold* (tipo *default*) com k=10 *folds*), como indica a **Figura 10**.

```
random_search = RandomizedSearchCV(classifier, param_distributions=hp_dist, n_iter=20, cv=10)
random_search.fit(features, target)
```

Figura 10: *Random Search* dos hiper-parâmetros a 20 iterações e *cross-validation* do tipo *KFold* com k=10

Por forma a visualizar os resultados de forma mais interessante, foi criada uma função auxiliar (**Figura 11**) que pega nos resultados do método *Random-SearchCV* e apresenta os N melhores modelos encontrados durante a pesquisa (por defeito, N=3), como indica a **Figura 12**.

```python
# Prints the top-X (by default = 3) results from model's hyperparameters optimization
def print_top_results(results, n_top=3):
    for i in range(1, n_top + 1):
        candidates = np.flatnonzero(results['rank_test_score'] == i)
        for candidate in candidates:
            print("Model with rank: {0}".format(i))
            print("Mean validation score: {0:.3f} (+/- {1:.3f})".format(results['mean_test_score'][candidate],
                                                                       results['std_test_score'][candidate]))
            print("Parameters: {0}\n".format(results['params'][candidate]))
```

Figura 11: Função de impressão dos N melhores modelos encontrados

```
Model with rank: 1
Mean validation score: 0.733 (+/- 0.099)
Parameters: {'nodes_per_layer': 16, 'learning_rate': 0.01, 'hidden_layers': 2, 'activation_fn': 'relu'}

Model with rank: 2
Mean validation score: 0.626 (+/- 0.140)
Parameters: {'nodes_per_layer': 10, 'learning_rate': 0.01, 'hidden_layers': 2, 'activation_fn': 'sigmoid'}

Model with rank: 3
Mean validation score: 0.532 (+/- 0.044)
Parameters: {'nodes_per_layer': 1, 'learning_rate': 1e-08, 'hidden_layers': 2, 'activation_fn': 'relu'}
```

Figura 12: Resultado correspondente aos três melhores modelos encontrados durante uma otimização do tipo *Random Search*

Por forma a obter os valores da *accuracy* para este tipo de otimização, foram feitos vários testes, como mostra a tabela presente na **Figura 13**.

RandomSearch (10 iterações, cv k=10)	accuracy	hidden_layers	nodes_per_layer	activation_fn	learning_rate
#	73.30%	2	16	ReLU	0.01
#	72.30%	8	14	ReLU	0.00576
#	72.43%	7	15	ReLU	0.0658
#	62.50%	8	7	Sigmoid	0.00724
#	73.14%	2	16	ReLU	0.00724

Figura 13: Tabela com resultados da otimização do tipo *Random Search*

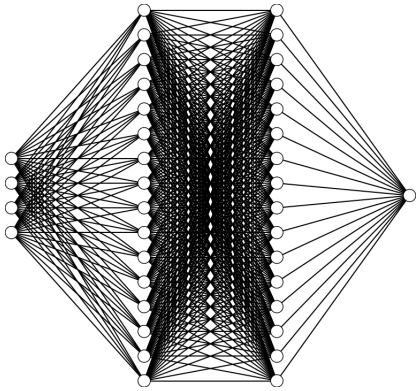

Figura 14: Rede Neuronal Artificial [4, 16, 16, 1]

8 Hiper-parametrização *Bayesian Optimization* (ficheiro *bayesian_opt.py*)

No entanto, o processo de hiper-parametrização utilizando *Random Search* é um método que seleciona os hiper-parâmetros de forma completamente aleatória, sem que seja utilizado algum mecanismo de pesquisa mais inteligente.

Assim sendo, o grupo procurou exemplos de implementações de outras formas de hiper-parametrização, como sendo os Algoritmos Genéticos, *Gaussian Process* ou Otimização Bayesiana. Acabámos por optar por este último pelo facto de termos encontrado a biblioteca *GPy* e *GPyOpt* que nos facilitam bastante este processo, possuindo inclusivamente alguns exemplos de aplicação.

Assim sendo, a nosso solução passou por criar uma classe para este *dataset*, denominada *MM* (de *Mammographic Masses*). Esta classe possui como variáveis de instância campos para os valores dos hiper-parâmetros, um campo para o modelo utilizado, e outro para alojar os dados lidos do *dataset*. Além disso, possui também uma função *mm_data* que se encarrega de carregar os dados e fazer o seu pré-processamento (como descrito nas secções anteriores), *mm_model* que consiste na criação do modelo (similar à das secções anteriores) e uma função de *fit* com mecanismo de *early stopping* e outra de *evaluate*. Todos estas funções são depois combinadas num único método, denominado *mm_run*, que recebe os valores individuais para cada um dos hiper-parâmetros (que em cada iteração lhe são fornecidos), fazendo retornando o valor da avaliação (*loss* e *accuracy*) após a chamada do *fit* e do *evaluate*.

Depois, o *run_mm* é chamado por uma função *f* que será passada ao método de otimização bayesiana. Esta função *f* apenas pega no resultado do *run_mm* e efetua a impressão na consola dos resultados intermédios de *loss* e *accuracy*, como mostra a **Figura 15**.

```
LOSS:   0.19901499146764928      ACCURACY:  0.7563636361468922
[0.19901499146764928, 0.7563636361468922]
[[2.00000000e+00 1.80000000e+01 1.00000000e+00 9.50270107e-04]]
```

Figura 15: Impressão dos valores intermédios de *loss* e *accuracy*

Finalmente, esta função é então passada ao método da biblioteca *GPyOpt*, denominado *BayesianOptimization*, juntamente com os intervalos de variação para cada um dos hiper-parâmetros. Por fim, é corrida a otimização, definindo um número máximo de iterações, sendo posteriormente imprimidos os resultados como apresenta a **Figura 16**. Nesta função é então permitido passar o valor da métrica que se pretende que sirva de base para a otimização (*loss* ou *accuracy*). A **Figura 17** possui um conjunto de resultados referentes à otimização da métrica de *loss* para uma otimização bayesiana com 100 iterações.

```
Optimized Parameters:
    hidden_layers:  8.0
    nodes_per_layer:    15.0
    activation_fn:  0.0
    learning_rate:  0.01

optimized accuracy: 0.7745454554124312
```

Figura 16: Exemplo de um resultado após otimização bayesiana

BayesianOpt (100 iterações)	loss	hidden_layers	nodes_per_layer	activation_fn	learning_rate
#	16.41%	4	18	ReLU	0.01
#	24.99%	4	15	ReLU	0.01
#	16.24%	4	13	ReLU	0.00931
#	16.76%	2	11	ReLU	0.01
#	17.42%	8	19	ReLU	0.00792

Figura 17: Tabela com resultados da otimização do tipo *Bayesian Optimization* (para métrica de *loss*)

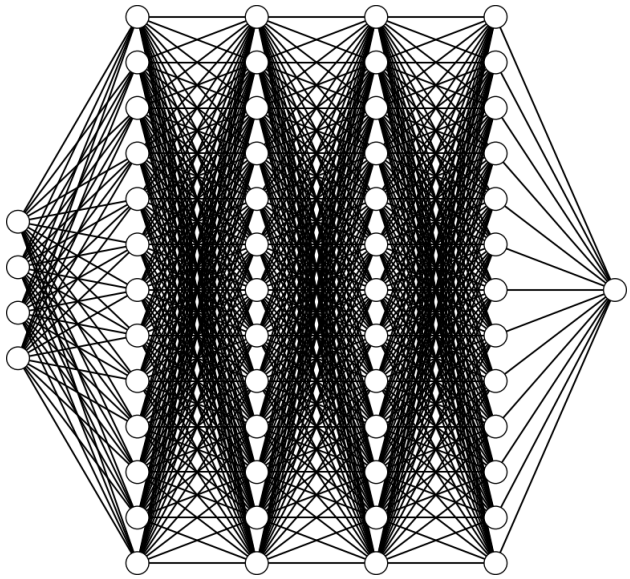

Figura 18: Rede Neuronal Artificial [4, 13, 13, 13, 13, 1]

No entanto, como vimos na fase de análise dos dados (i.e. fase de *data visualization*), os valores do *target* encontram-se quase igualmente distribuídos pelas classes existentes (0 ou 1, benigno ou maligno). Assim sendo, a métrica mais indicada para usar como base para a otimização seria a da *accuracy*. Desta forma, foi feito um novo conjunto de testes de otimização bayesiana tendo a métrica de *accuracy* como base da otimização (100 iterações), presente na **Figura 19**.

BayesianOpt (100 iterações)	accuracy	hidden_layers	nodes_per_layer	activation_fn	learning_rate
#	77.82%	2	4	ReLU	0.00891
#	78.18%	4	14	ReLU	0.00783
#	77.45%	2	18	ReLU	0.00964
#	77.82%	2	17	ReLU	0.00806
#	77.82%	8	18	ReLU	0.0079

Figura 19: Tabela com resultados da otimização do tipo *Bayesian Optimization* (para métrica de *accuracy*)

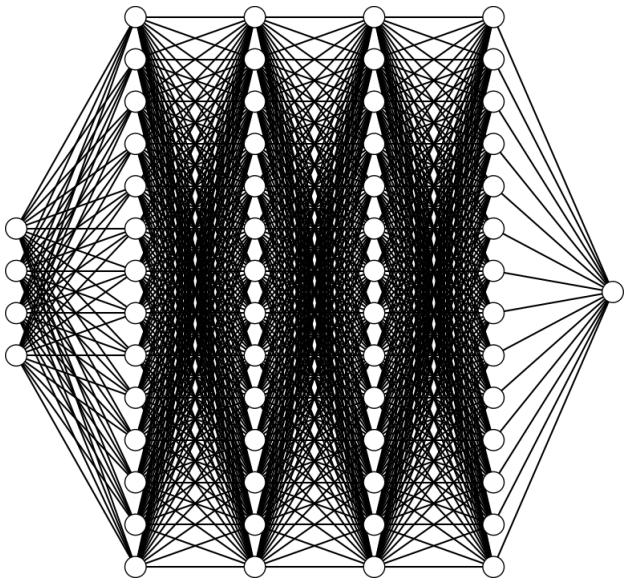

Figura 20: Rede Neuronal Artificial [4, 14, 14, 14, 14, 1]

9 Conclusões e Trabalho Futuro

Após a sua finalização o grupo considera que fez um bom trabalho ao aplicar as técnicas desenvolvidas ao longo das aulas ao preparar e analisar os dados e ao adaptar-se às dificuldades encontradas na utilização das técnicas de *cross-validation* e na criação da rede neuronal ao utilizar a biblioteca *Keras* de forma a tornar todo o processo mais simples.

Para trabalho futuro seria interessante implementar uma opção de definir os números de nós por cada camada individual da rede neuronal e utilizar um outro tipo de otimização mais complexo, como a otimização evolucionária (algoritmos genéticos), como complemento à otimização bayesiana realizado para a tarefa extra do enunciado.

10 Anexo

10.1 Gráficos de distribuições das métricas do *dataset*

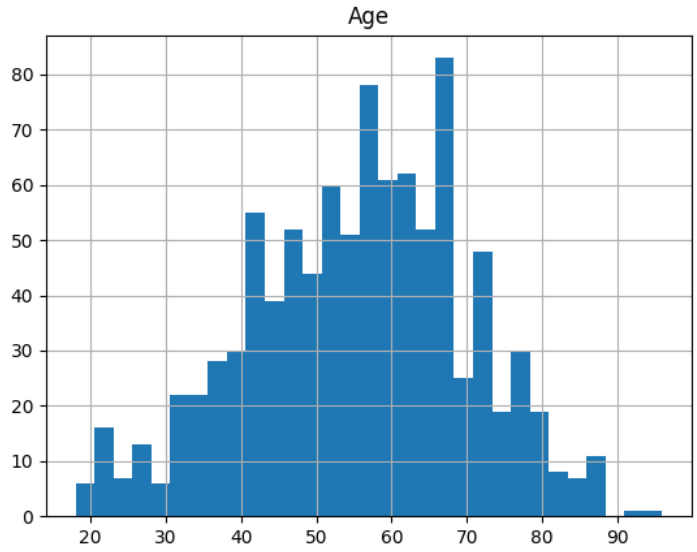

Figura 21: Gráfico de distribuição de *Age*

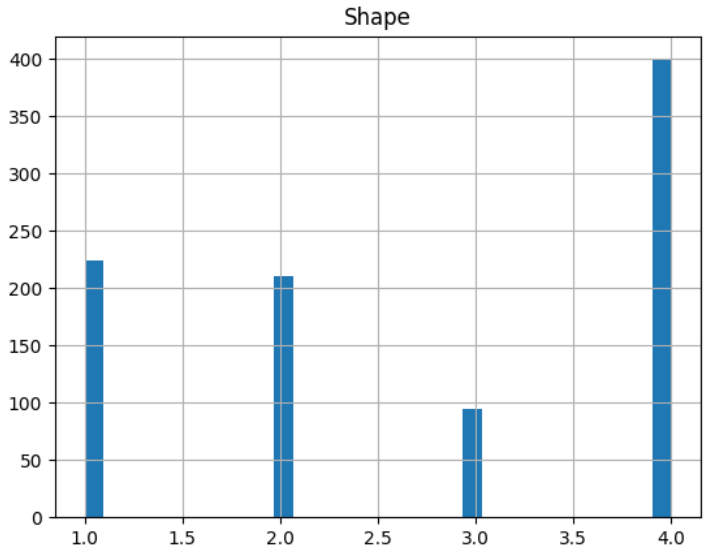

Figura 22: Gráfico de distribuição de *Shape*

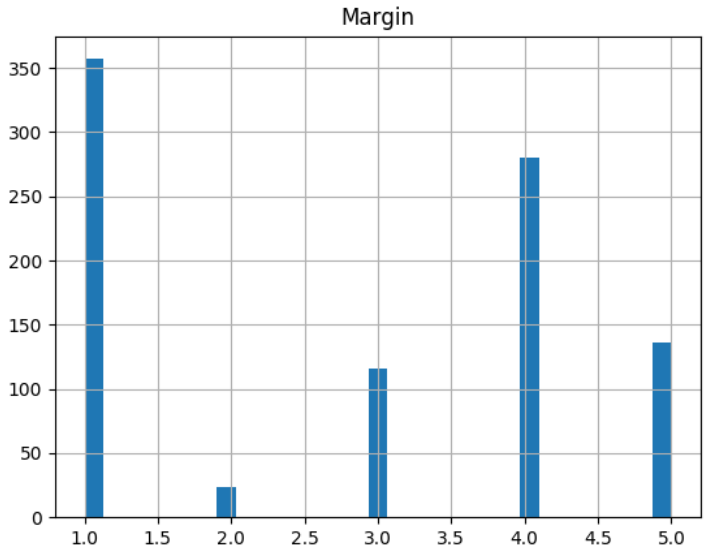

Figura 23: Gráfico de distribuição de *Margin*

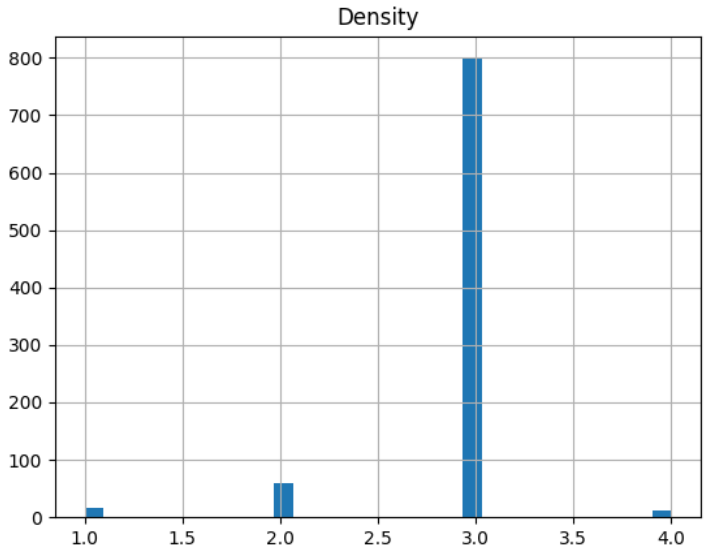

Figura 24: Gráfico de distribuição de *Density*

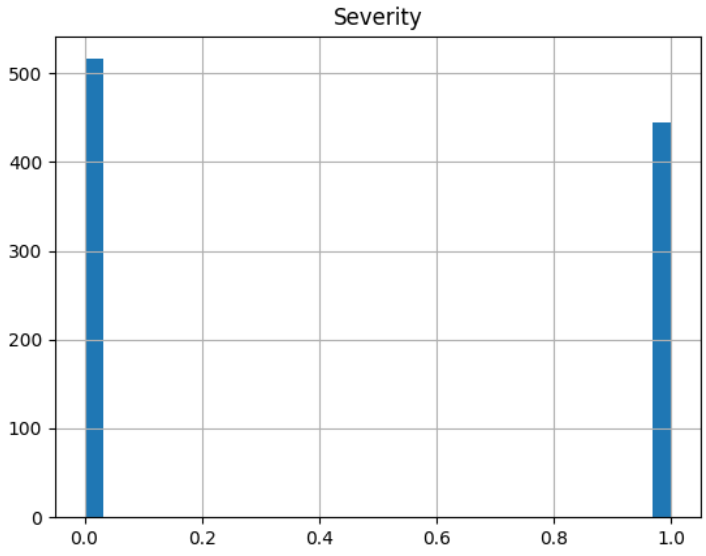

Figura 25: Gráfico de distribuição de *Severity*