Comportamento Biométrico de Alunos durante Exames

Estudo do *dataset* utilizando *scikit-learn*

Grupo 6

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

Dezembro 2018

Universidade do Minho
Escola de Engenharia

Trabalho Prático 2
Sistemas Inteligentes – Aprendizagem e Extração de Conhecimento
MEI / MIEI – Universidade do Minho

1 Introdução

O presente trabalho surge no âmbito da Unidade Curricular de Aprendizagem e Extração de Conhecimento e tem como objetivo o estudo e manipulação de um *dataset* segunda as técnicas de *Machine Learning* disponibilizadas pelas biblioteca *scikit-learn* da linguagem de programação *Python*.

O presente relatório encontra-se dividido nas etapas que foram percorridas até ao resultado final, procurando explicar detalhadamente todo o processo, bem como as decisões tomadas. Na secção final, faz-se uma análise crítica dos resultados obtidos e do trabalho realizado.

O relatório faz referência ao código entregue, tanto aos ficheiros na raiz do mesmo, como aos ficheiros de teste que foram usados para fazer comparações e tirar conclusões adicionais e que estão presentes na pasta "teste".

2 Data Acquisition

Esta fase não foi contemplada durante este trabalho prático visto que nos foram fornecidos os *datasets* com os quais teríamos de trabalhar.

Os *datasets* em causa dizem respeito à análise dos dados biométricos de alunos universitários durante situações de exame. No caso de um desses *datasets*, os dados biométricos correspondem a dados relativos ao comportamentos dos alunos na tomada de decisões durante os exames. No outro, são referentes ao uso do rato.

Em ambos os casos, o objetivo passa por tratar os dados procurando encontrar uma relação entre os mesmos e um resultado numérico (*"PSS Stress"*) que identifica o nível de *stress* percecionado pelos alunos.

No caso do nosso grupo, coube-nos tratar o *dataset* correspondente à análise dos dados biométricos dos alunos relativos ao uso do rato.

3 Data Visualization (ficheiro visualization.py)

Esta etapa diz respeito à visualização dos gráficos de distribuição das *features* do nosso *dataset*. Para isso, a nossa aplicação percorre as colunas do dataset, criando uma imagem chamada *"dist-FEATURE"* em que *FEATURE* é o nome de cada uma das colunas do *dataset*. Estas imagens são criadas na pasta *"dist"* da raíz do projeto. Através do código abaixo – que tira partido da biblioteca *matplotlib* –, é possível gerar as ditas imagens:

```
mkdir("dist")

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

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

Assim sendo, são geradas as imagens das distribuições que se encontram em anexo.

Por forma a termos uma visão mais completa dos valores que caracterizam as distribuições das *features*, foi também chamada a função *describe()* para o nosso *Pandas dataset*, que nos permitiu construir a seguinte tabela da figura seguinte.

	PSS_Stress	absoluteSum	averageDista	averageExces	clickDurations	distanceBetweenC	distanceDuringCl
count	335.00	329.00	327.00	329.00	333.00	327.00	325.00
mean	25.00	10037.11	80.37	9.53E+12	215.84	449.45	95.92
std	6.87	5269.02	32.69	2.81E+13	109.22	219.12	87.90
min	4.00	34.99	2.81	1.00	14.85	5.00	1.00
25%	21.00	6312.73	57.25	4.39	145.39	314.91	39.41
50%	25.00	9256.85	74.33	6.71	198.28	394.32	67.53
75%	29.00	13509.08	96.30	11.98	258.97	534.62	128.90
max	50.00	29630.74	281.27	9.22E+13	897.98	1940.55	767.88

Figura 1: Estudo dos valores das *features* do *dataset* (e *PSS_Stress*) (1)

	distancePointerToL	excessOfDist	mouseAccel	mouseVel	signedSumofD	timeBetweenCl	timeDoubleCl
count	327.00	328.00	335.00	335.00	329.00	326.00	315.00
mean	41903.90	859.09	0.84	1.01	10.25	11924.98	350.61
std	36522.31	617.40	0.22	3.22	21.67	4232.05	66.47
min	11.22	2.85	-0.64	0.42	-117.08	2480.09	1.00
25%	16287.07	442.39	0.71	0.67	-0.85	8797.31	318.89
50%	29052.77	675.44	0.81	0.77	8.90	11382.71	359.00
75%	57957.54	1099.42	0.94	0.90	18.45	14214.83	394.50
max	224925.90	5191.31	1.90	58.49	180.00	28231.71	495.00

Figura 2: Estudo dos valores das *features* do *dataset* (2)

Como podemos constatar pela tabela acima, as *features* possuem uma distribuição de valores em intervalos e escalas bastante diferentes, pelo que será necessário normalizar os seus valores para um estudo mais eficaz.

Assim, por forma a podermos comparar os gráficos das distribuições, iremos gerar as figuras como descrito acima, mas fazendo primeiro a normalização dos valores das colunas para uma escala de 0 a 1.

Desta forma, podemos ainda tentar comparar as distribuições das *features* (cujas imagens não normalizadas se encontram em anexo) com a distribuição da coluna *PSS_Stress*, presente na seguinte figura:

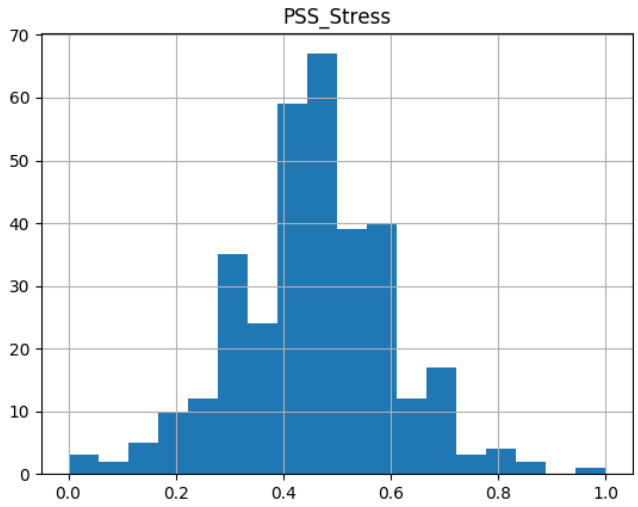

Figura 3: Distribuição de *PSS_Stress* (*target*)

À primeira vista, parece-nos que as colunas do dataset que possuem gráficos de distribuição mais semelhantes são: *mouseAcceleration*, *timeBetweenClicks*, *signedSumofDegreesBetweenClicks* e *timeDoubleClicks*. Esta informação poderá vir a revelar-se útil durante o processo de *Feature Selection*.

4 Data Preprocessing

4.1 Missing Data Filtering (ficheiro missing_data.py)

Ao analisarmos o *dataset*, reparámos que haviam alguns registos que possuíam valores do tipo 'NaN'. Estes casos são bastante comuns e podem ter origens variadas tanto em observações que, por qualquer motivo, não foram registadas, como em casos em que a informação do *dataset* se encontra corrompida.

Independentemente disso, coube-nos a tarefa de tratar destes dados em falta. Uma das estratégias seria descartar por completo as linhas (registos) que possuíssem pelo menos um valor do tipo 'NaN'. Em alternativa, poderíamos substituir os valores 'NaN' "artificialmente"pelo valor da média da coluna referente à *feature* correspondente.

De forma a podermos escolher mais acertadamente, precisávamos de saber

a dimensão que esta falta de valores tinha no *dataset*. Tal foi possível através do seguinte código que nos deu o número de linhas que possuem pelo menos um valor do tipo 'NaN':

```
print(data.isnull().any(axis=1).sum())
```

Assim, obtivemos o valor de 20 linhas que possuem pelo menos um valor 'NaN', o que perfaz cerca de 6% de todos os registos do *dataset*.

Apesar deste valor não ser demasiado elevado, optámos por não perder nenhuma informação, substituindo os 'NaN' pelo valor da mediana. Escolhemos a mediana porque a partir da análise feita na secção *Data Visualization*, foi possível constatar que haviam bastantes *outliers*, pelo que não queríamos que os valores dos dados fossem tão distorcidos pelos mesmos.

4.2 Feature Selection (ficheiro feature_selection.py)

Como forma de podermos saber quais as colunas de *input* mais relevantes para o valor do *output* (*PSS_Stress*), foi efetuado o processo de *feature selection*.

Para isso, utilizámos a função *SelectKBest* (*Unvariate Selection*) para determinar quais seriam essas colunas. Depois, para que fosse mais percetível essa informação, criou-se um ciclo *for* auxiliar que apenas imprimisse quais as colunas selecionadas (índices e respetivos nomes).

No entanto, faltou ainda saber qual o número de *features* a utilizar na função *SelectKBest* (parâmetro K). Para isso, foi feito uma comparação dos valores de accuracy para valores de K entre 1 e 10. Para isso, utilizou-se um modelo SVC *default* (i.e. sem parâmetros), uma standardização dos dados para uma escala de 0 a 1, e cálculo do resultado com 25% dos dados a serem utilizados para teste (sem *cross-validation*). O resultado foi o seguinte:

SelectKBest	Accuracy (SVC default + normalização [0..1] + 25% testes)	Colunas escolhidas
k = 1	7.143	10
k = 2	6.746	3,10
k = 3	5.555	3,6,10
k = 4	6.349	3,4,6,10
k = 5	7.539	3,4,6,7,10
k = 6	5.159	1,3,4,6,7,10
k = 7	7.143	1,2,3,4,6,7,10
k = 8	7.143	1,2,3,4,6,7,8,10
k = 9	6.349	1,2,3,4,6,7,8,10,11
k = 10	6.349	1,2,3,4,6,7,8,10,11,12
k = 11	4.762	0,1,2,3,4,6,7,8,10,11,12
k = 12	4.762	0,1,2,3,4,5,6,7,8,10,11,12
k = 13	5.952	0,1,2,3,4,5,6,7,8,9,10,11,12

Figura 4: Comparação *feature selection*

Assim sendo, pudemos concluir que o melhor seria utilizar um K com o valor de 5. É de notar, no entanto, que os valores do resultado apresentaram uma grande variância entre eles (i.e. para um mesmo valor de K), pelo que os resultados apresentados na tabela correspondem à média de três tentativas para cada K.

Ambos os processos descritos anteriormente, encontram-se nas duas figuras seguintes:

```
selector = SelectKBest(chi2, k=5)
selector.fit(data, target)
cols = selector.get_support(indices=True)
cols_names = list(data.columns[cols])

for idx, (ci, cn) in enumerate(zip(cols, cols_names)):
    print("*" * (len(cols) - idx) + " " * idx, ci, cn)

data = data[cols_names]
```

Figura 5: Processo de *feature selection*

```
***** 3 clickDurations
****  4 distanceBetweenClicks
***   6 distancePointerToLineBetweenClicks
**    7 excessOfDistanceBetweenClicks
*     10 signedSumofDegreesBetweenClicks
```

Figura 6: Impressão de um resultado do *feature selection*

4.3 Normalization/Standardization (ficheiros normalization_compare.py / standardization_compare.py)

De modo a se poder obter melhores resultados deve-se diminuir a escala dos dados, especialmente se certos atributos possuírem valores demasiado elevados. Para isto usaram-se as técnicas de *normalization* e de *standardization*.

Na técnica de *normalization* os dados são convertidos de forma a que sua norma (l1 ou l2) seja igual a 1. O processo efetuado foi o seguinte:

```
normalizer = preprocessing.Normalizer(norm='l1')
values_normalized = normalizer.transform(data.values)
data = pd.DataFrame(values_normalized, columns=data.columns)
```

Figura 7: Processo de *normalization*

Por forma a percebermos qual seria o modelo de normalização mais adequado (l1 ou l2), foram feitos cálculos da *accuracy* para um modelo SVC default (i.e. sem parâmetros), *feature selection* do tipo *SelectKBest* com k=5 (por ter apresentado o melhor valor na secção anterior) e com 25% dos dados utilizados para teste (sem *cross-validation*). O resultado foi o seguinte:

	Accuracy (SVC default + selectkbest(k=5) + 25% testes)
L1 normalization	5.952
L2 normalization	5.952

Figura 8: Comparação de *normalization*

Como os resultados obtidos foram iguais, poderíamos ter utilizado qualquer uma das duas, mas optámos pela normalização do tipo l1.

Relativamente ao processo de *standardization*, após uma consulta da *API*, verificou-se que ao invés de se usar o processo de *standardization* mais comum (*Min Max Standardization*), deveria-se usar um processo denominado de *Robust Standardization*. Este processo permite um melhor ajuste quando se trata de dados que contêm *outliers*, que é o caso do *dataset* em estudo.

No entanto, optámos por fazer o teste por forma a comparar este *RobustScaler* com o *MinMaxScaler* para um intervalo entre 0 e 1. O resultado foi o seguinte:

	Accuracy (SVC default + selectkbest(k=5) + 25% testes)
MinMaxScaler(0,1)	6.349
RobustScaler	7.54

Figura 9: Comparação de *standardization*

Assim sendo, pudemos comprovar a informação presente da documentação do *scikit-learn*, passando a utilizar o *RobustScaler* para a standardização. Este processo foi, então, realizado da seguinte forma:

```
robust_scaler = preprocessing.RobustScaler()
values_standardized = robust_scaler.fit_transform(data.values)
data = pd.DataFrame(values_standardized, columns=data.columns)
```

Figura 10: Processo de *standardization*

5 Model Selection, Model Training and Validation (ficheiros cv_compare.py e model.py)

Tendo-se já efetuado os processos de tratamento de dados, *normalization* e *standardization*, assim como a seleção dos atributos mais significativos, procedeu-se à seleção do melhor modelo.

Para isso, foi necessário testar vários modelos diferentes utilizando a técnica de *cross-validation* (em vez do teste direto).

No que toca ao *cross-validation*, testámos dois métodos: *K-Fold* e *ShuffleSplit*. Fizemo-lo para vários valores relativos aos *splits* (de 1 a 10). Os resultados foram os seguintes:

	Accuracy KFold CV (%)	Accuracy ShuffleSplit CV (%)
k = 1	-	3.529
k = 2	6.106	2.941
k = 3	7.842	2.288
k = 4	7.364	3.922
k = 5	8.206	3.922
k = 6	7.127	4.248
k = 7	8.161	6.303
k = 8	7.442	4.412
k = 9	8.121	5.447
k = 10	6.69	3.627
SVC default, SelectKBest (k=5), Standardization (RobustScaler)		

Figura 11: *K-Fold CV* vs. *ShuffleSplit CV* (splits = 1..10)

Assim sendo, pudemos constatar que o melhor resultado foi obtido para a técninca de *K-Fold cross-validation* para 5 *splits*.

Posto isto, restou-nos testar os resultados para vários modelos, por forma a selecionar o melhor. Optou-se por testar os modelos de *Nearest Neighbors*, *Support Vector Classification*, *Gaussian Process*, *Decision Tree*, *Random Forest*, *Neural Network*, *AdaBoost* e *Naive Bayes*.

Para cada um destes modelos foram testados os resultados produzidos através da *normalization* (l1) e da *standardization* (*RobustScaler*) (analisados anteriormente) dos dados. Foi também escolhido o melhor método de *feature selection* visto anteriormente: *SelectKBest* com k = 5.

O resultado obtido foi o seguinte:

```
***** 3 clickDurations
****  4 distanceBetweenClicks
***   6 distancePointerToLineBetweenClicks
**    7 excessOfDistanceBetweenClicks
*     10 signedSumofDegreesBetweenClicks
 ### Normalization ###
Nearest Neighbors Accuracy: 0.039258 (+/- 0.020391)
SVM Accuracy: 0.077462 (+/- 0.033696)
Gaussian Process Accuracy: 0.056381 (+/- 0.051491)
Decision Tree Accuracy: 0.049708 (+/- 0.032751)
Random Forest Accuracy: 0.034748 (+/- 0.028303)
Neural Net Accuracy: 0.060194 (+/- 0.033275)
AdaBoost Accuracy: 0.076939 (+/- 0.056166)
Naive Bayes Accuracy: 0.053071 (+/- 0.041586)

### Standardization ###
Nearest Neighbors Accuracy: 0.014245 (+/- 0.017349)
SVM Accuracy: 0.082059 (+/- 0.052744)
Gaussian Process Accuracy: 0.045302 (+/- 0.053427)
Decision Tree Accuracy: 0.047322 (+/- 0.090372)
Random Forest Accuracy: 0.063199 (+/- 0.047135)
Neural Net Accuracy: 0.074308 (+/- 0.064457)
AdaBoost Accuracy: 0.060209 (+/- 0.046526)
Naive Bayes Accuracy: 0.028006 (+/- 0.054382)
```

Figura 12: Resultados produzidos através dos diferentes modelos

Daqui conclui-se que a melhor opção será proceder ao processo de *standardization* dos dados e efetuar o modelo de *Support Vector Classification*, que se traduz numa *accuracy* de 8.2059%.

6 Hyperparameter Optimization (ficheiro optimization.py)

Tal como falado na secção anterior, os melhores resultados foram atingidos através de uma técnica de *standardization* dos dados aplicados a um modelo de *Support Vector Classification*. Como os resultados obtidos foram atingidos através dos valores dos parâmetros por defeito deste modelo, optou-se por fazer uma hiper-parametrização destes parâmetros de modo a se conseguir atingir uma melhor *accuracy*.

Para uma melhor organização deste processo, optou-se por criar um novo *script*, no qual foi efetuada uma replicação do tratamento dos dados provenientes

do *dataset*, da seleção dos atributos mais significativos e da *standartization* dos dados. Depois de se efetuar todo este processo, fez-se um estudo de todos os parâmetros envolvidos no modelo de *Support Vector Classification*. De seguida é apresentada uma lista de todos estes parâmetros. Nessa lista apresenta-se também, uma indicação dos parâmetros que não foram selecionados para se efetuar o *tuning* do modelo, pelo facto de não se mostrarem relevantes.

- ***C***: Valor do tipo real que traduz o nível de penalidade.
- ***kernel***: Tipo de *kernel* usado no algoritmo do tipo *string*. Apenas pode ser *linear*, *poly*, *rbf* ou *sigmoid*.
- ***degree***: Valor do tipo inteiro apenas aplicado ao *kernel poly*.
- ***gamma***: Valor do tipo real apenas aplicado aos *kernels rbf*, *poly* e *sigmoid*.
- ***coef0***: Valor do tipo real que traduz o termo independente da função do *kernel* selecionado. Apenas relevante para os *kernels poly* e *sigmoid*.
- ***shrinking***: Valor booleano que indica a utilização de uma heurística de encolhimento.
- ***probability***: Valor booleano que indica a utilização de estimações baseadas em probabilidades.
- ***tol***: Valor do tipo real que indica a tolerância.
- ***cache_size***: Valor real que indica a o tamanho em *MB* usado para a *cache* do *kernel*. Este valor não foi estimado por não se revelar importante.
- ***class_weight***: Recebe um dicionário que indica os pesos das diferentes classes envolvidas neste modelo. Este valor não foi estimado por não se conseguir encontrar novas soluções para este parâmetro.
- ***verbose***: Valor booleano que apenas indica se todas as decisões tomadas pelo modelo devem ser apresentadas como *output*, pelo que não foi estimado.
- ***max_iter***: Valor inteiro que indica o número máximo de iterações permitidas. Esteve valor não foi estimado visto que o seu valor por defeito indica que não há qualquer tipo de restrições ao número máximo de iterações.
- ***decision_function***: Indica o tipo de função de decisão usada (*ovo* ou *ovr*).
- ***random_state***: *Seed* dada para a geração de um valor aleatório usado para calcular as probabilidades. Este valor não foi estimado.

Após o estudo de todos os parâmetros indicados acima e dos seus tipos, procedeu-se então ao *tuning* do modelo. De modo a se poder observar quais as melhores combinações dos parâmetros criou-se uma função auxiliar que indica os melhores valores selecionados dos parâmetros e respetiva *accuracy*. Assim, a visualização dos resultados dos melhores *scores* tornou-se bastante mais simples. A função é a seguinte:

```
def report(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 13: Função que traduz os melhores resultados

Para se efetuar a hiper-parametrização construiu-se um dicionário que continha como chaves os nomes dos parâmetros que se pretendia estimar e como valores a gama de valores a que deveriam ser atribuídos a esses parâmetros. Para este processo foi efetuada a técnica de *Randomized Parameter Optimization*. O processo efetuado foi o seguinte:

```
clf_model = SVC()

param_dist = {'C': np.random.uniform(low=0.0, high=2.0, size=(10,)),
              'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
              'degree': ss.randint(1, 5),
              'gamma': ['auto', 'scale'],
              'coef0': np.random.uniform(low=0.0, high=2.0, size=(10,)),
              'shrinking': [True, False],
              'probability': [True, False],
              'tol': np.random.uniform(low=0.0, high=2.0, size=(10,)),
              'decision_function_shape': ['ovo', 'ovr']}

rs = RandomizedSearchCV(clf_model, param_distributions=param_dist, n_iter=50, cv=5)
rs.fit(data, target)
report(rs.cv_results_)
```

Figura 14: Processo efetuado para a hiper-parametrização do modelo

Para o cálculo do resultado, foram utilizados os melhores métodos testados anteriormente: modelo SVC, *feature selection* do tipo *SelectKBest* com k=5, standardização dos dados com *RobustScaler* e *K-Fold cross-validation* com *splits*

= 5. A partir deste processo e, com recurso à função referida anteriormente, conseguiu-se o seguinte resultado final que levou a uma *accuracy* de 8.4%:

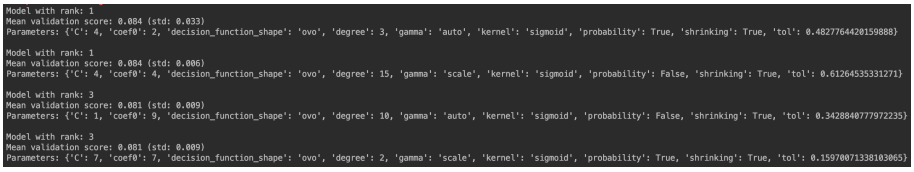

Figura 15: Resultado final da hiper-parametrização do modelo

7 Outra otimizações

7.1 Feature Selection (ficheiro optimization.py)

Nesta fase, o grupo tentou várias otimizações. Uma delas foi motivada pelo facto de que a função *SelectKBest* do processo de *feature selection* não estava a selecionar as *features* cuja distribuição tínhamos constatado que se parecia mais à da coluna de *output*, na fase de *Data Visualization*. Assim sendo, o grupo tentou passar várias outra funções à *SelectKBest* em vez da *f_classif* (e.g. *chi2*, *f_regression*, etc.), o que não surtiu os efeitos que queríamos.

Assim sendo, optámos por fazer um *feature selection* "manual", testando os resultado da hiper-parametrização do modelo para todas as combinações das colunas cuja distribuição nos pareceu mais semelhante à do *PSS_Stress*: *mouseAcceleration*, *timeBetweenClicks*, *signedSumofDegreesBetweenClicks* e *timeDoubleClicks*.

A partir deste processo, obteve-se o melhor resultado para a combinação de duas colunas: *mouseAcceleration* e *timeBetweenClicks*. Assim, conseguiu-se atingir – como indica a figura seguinte –, uma *accuracy* de 11.6%, bastante superior aos 8.4% conseguidos com a hiper-parametrização anterior, bem como aos 8.2% conseguidos na primeira fase (sem otimizações).

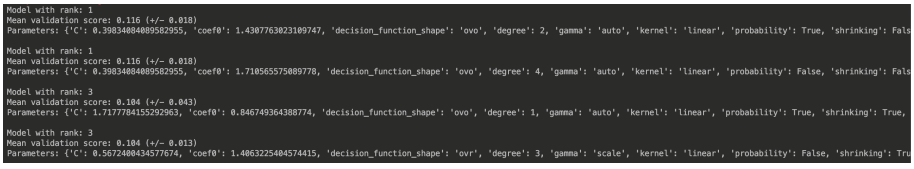

Figura 16: Resultado final da hiper-parametrização do modelo (com *feature selection* manual após estudo das distribuições)

7.2 Discretização da coluna de *output*, *PSS_Stress* (ficheiro discretization.py)

Após tudo o que foi feito anteriormente, o grupo percebeu a razão pela qual os resultados estavam a ser tão baixos: os nossos modelos estavam a tentar "adivinhar"valores para o *output* sendo que este estava numa escala muito grande. Na verdade, o *PSS_Stress* esteve sempre entre uma escala de 0 a 52, o que tornava difícil o acerto dos nossos modelos.

Assim sendo, normalizar também a coluna do *output* contribuiria para um maior acerto dos nossos modelos. No entanto, o mais indicado seria fazer uma discretização da coluna de *output*, *PSS_Stress*, agrupando os seus valores num número restrito de classes.

Para isso, bastaria chamar a função *KBinsDiscretizer*:

```
# Discretizar coluna de output (PSS_Stress)

target = np.array(target)
enc = KBinsDiscretizer(n_bins=5, strategy='quantile')
enc.fit(target.reshape(-1, 1))
target = enc.transform(target.reshape(-1, 1))
target = np.ravel(target)
```

Figura 17: Exemplo de discretização

Nessa função, *nbins* indica o número de classes a serem utilizadas e *strategy='quantile'* indica que todas classe devem ter um número igual de pontos.

Desta forma, poderíamos voltar a testar os nossos modelos para um número variável de classes. Os resultados foram os seguintes:

	Accuracy
nbins = 2	54.3
nbins = 3	37.6
nbins = 4	29.9
nbins = 5	23.6
SVC com hiper-parametrização, fs manual (colunas 9 e 12), standardização (RobustScaler) e 5-Fold CV	

Figura 18: Resultados discretização

8 Conclusões e Trabalho Futuro

Após a sua finalização, o grupo considera que fez um bom trabalho visto que aplicou todas as técnicas abordadas durante as aulas, bem como teve uma visão crítica sobre os resultados que foram aparecendo, nomeadamente ao nível da *feature selection* e da análise dos gráficos de distribuição.

Em retrospetiva, teria sido bastante vantajoso termos começado por discretizar a coluna do *output* (*PSS_Stress*) por forma a obtermos melhores resultados e, possivelmente, reduzirmos a enorme variância observada para os modelos anteriores.

9 Anexos

9.1 Gráficos da distribuição das *features* do *dataset*

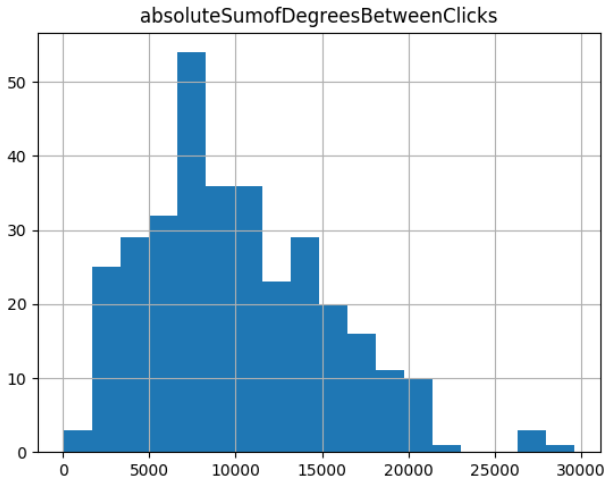

Figura 19: Distribuição de *absoluteSumofDegreesBetweenClicks*

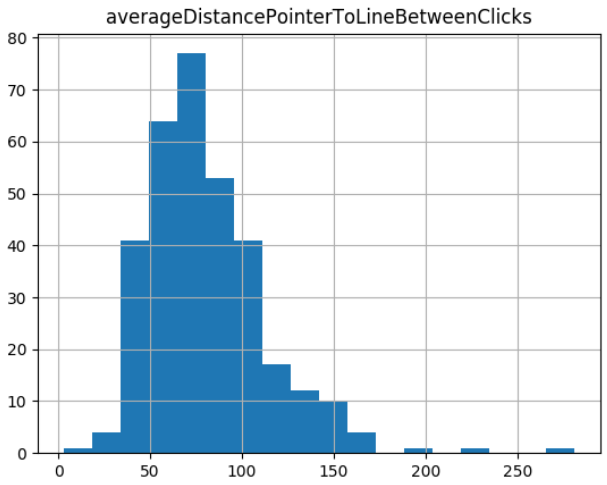

Figura 20: Distribuição de *averageDistancePointerToLineBetweenClicks*

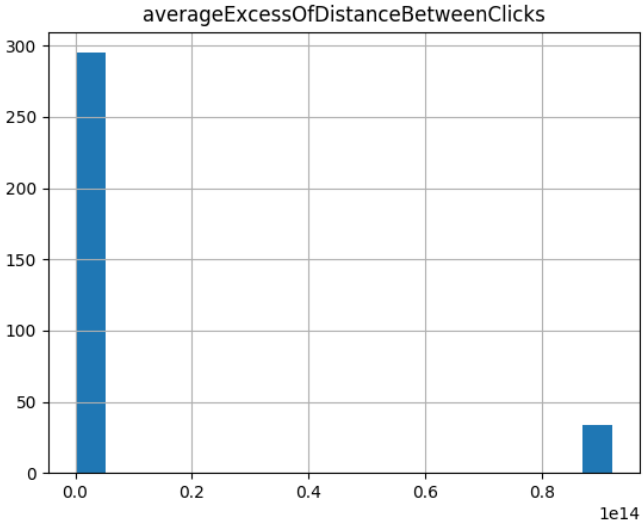

Figura 21: Distribuição de *averageExcessOfDistanceBetweenClicks*

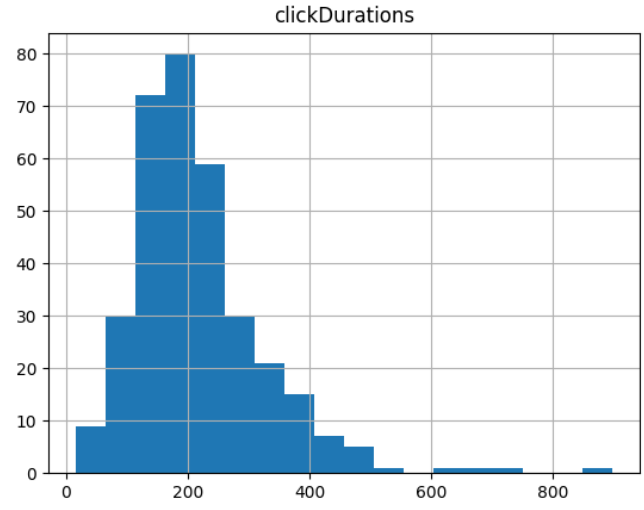

Figura 22: Distribuição de *clickDurations*

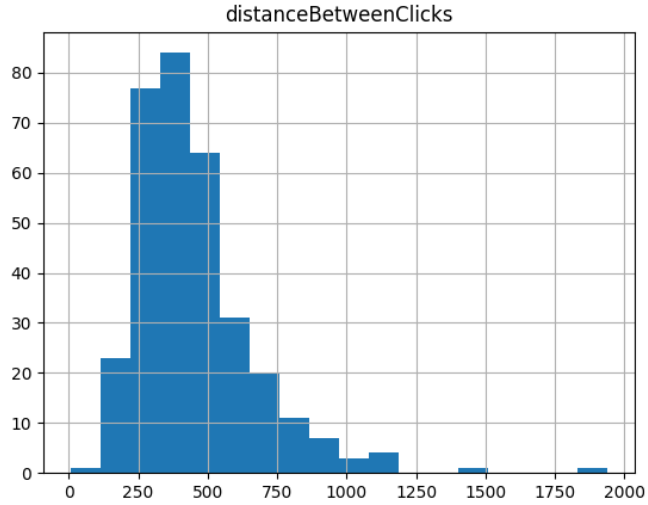

Figura 23: Distribuição de *distanceBetweenClicks*

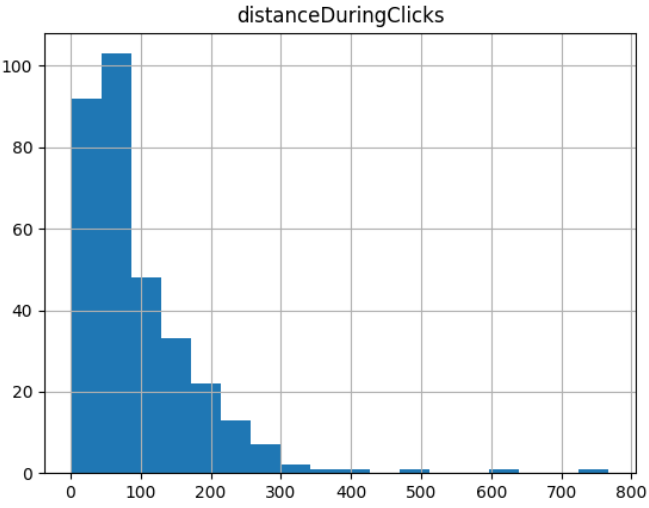

Figura 24: Distribuição de *distanceDuringClicks*

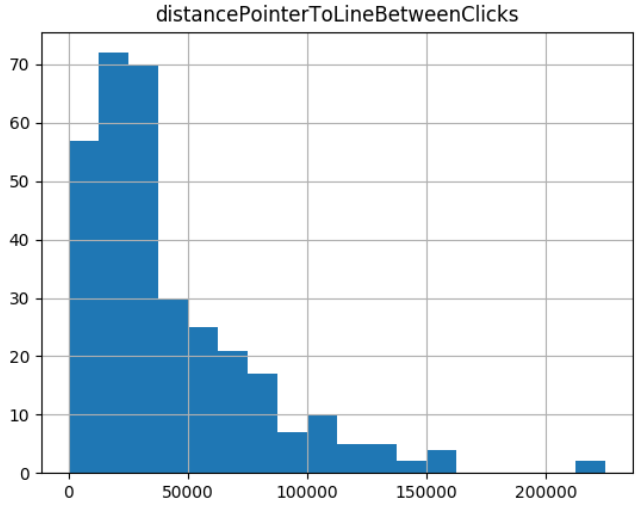

Figura 25: Distribuição de *distancePointerToLineBetweenClicks*

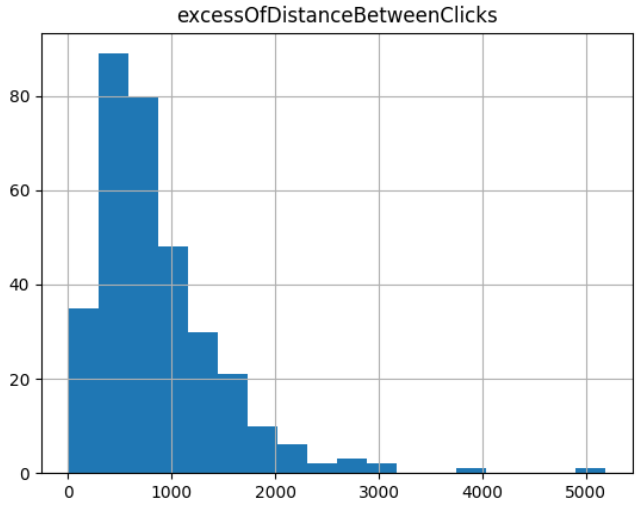

Figura 26: Distribuição de *excessOfDistanceBetweenClicks*

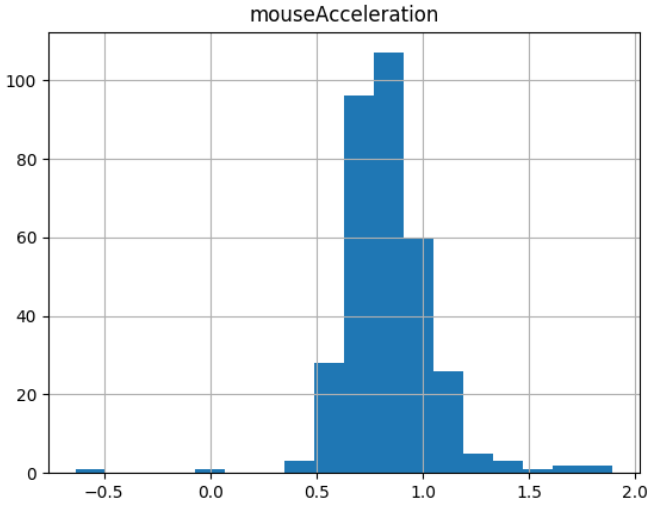

Figura 27: Distribuição de *mouseAcceleration*

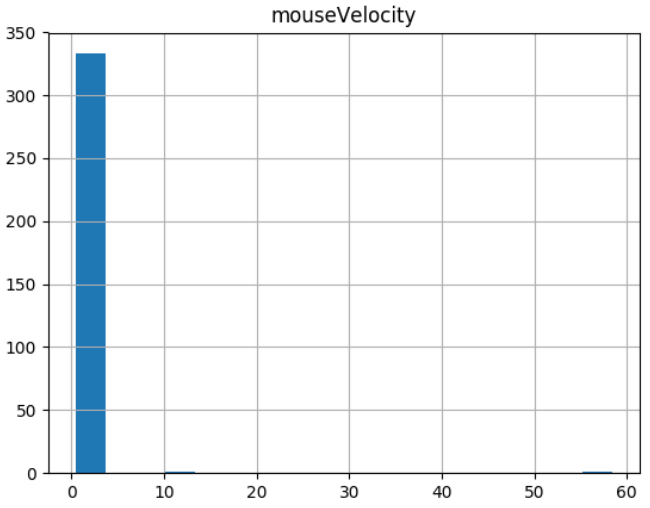

Figura 28: Distribuição de *mouseVelocity*

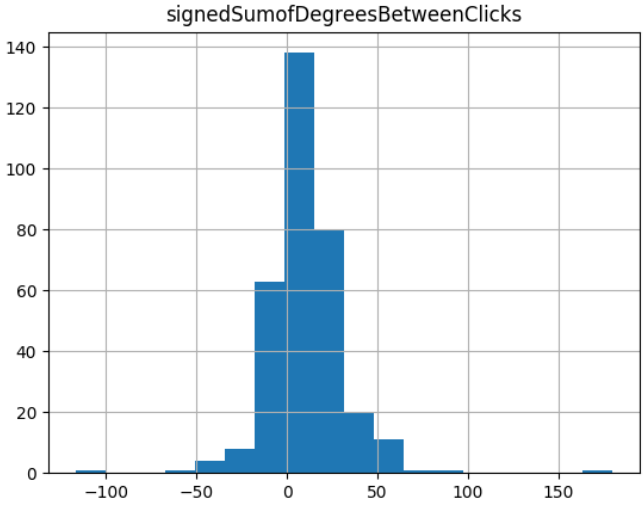

Figura 29: Distribuição de *signedSumofDegreesBetweenClicks*

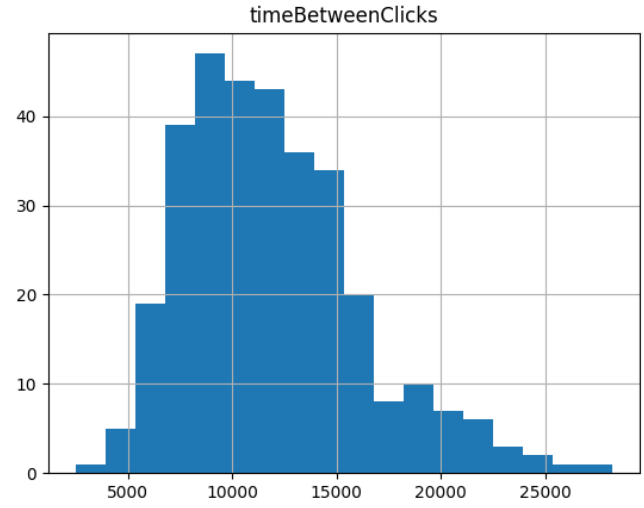

Figura 30: Distribuição de *timeBetweenClicks*

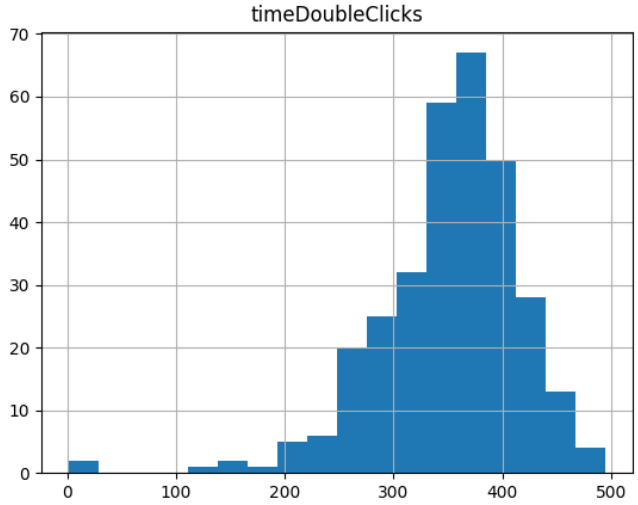

Figura 31: Distribuição de *timeDoubleClicks*