# 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.

## 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

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.

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

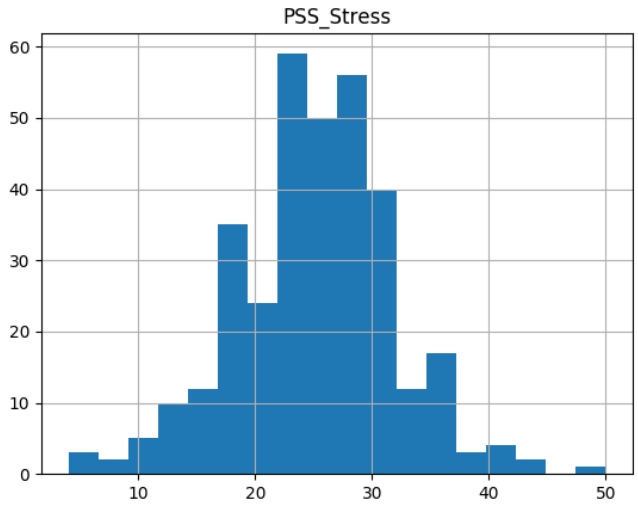


Figura 1: Distribuição de *PSS_Stress* (*target*)

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

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. Existem várias estratégias para o fazer, sendo talvez a mais comum a que consiste em substituir os valores em falta pela média dos valores das observações registadas. No entanto, como em praticamente todas as distribuições observadas na secção anterior existem *outliers*, optámos por utilizar a mediana em vez da média para obtermos valores mais realistas (i.e. menos distorcidos pelos

*outliers*).

## 4.2 Feature Selection

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 selecionas (índices e respetivos nomes). Ambos os processos 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 2: Processo de *feature selection*

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

Figura 3: Impressão de um resultado do *feature selection*

## 4.3 Normalization/Standardization

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 4: Processo de *normalization*

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. Este processo foi 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 5: Processo de *standardization*

# 5 Model Selection, Model Training and Validation

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. Optou-se por efetuar uma tentativa para 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* e da *standardization* dos dados.

Para isso, o treino dos modelos foi feito com recurso à técnina de *K-Fold Cross Validation* para 5 *folds*. O resultado 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 6: 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

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 7: 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 8: Processo efetuado para a hiper-parametrização do modelo

Sendo que 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%:

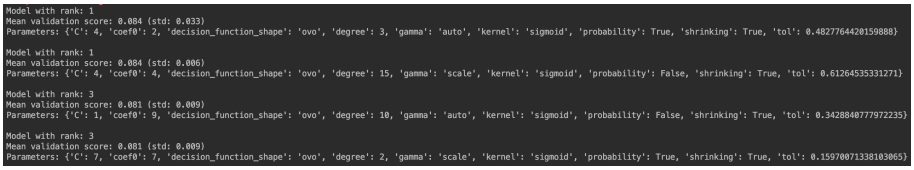


Figura 9: Resultado final da hiper-parametrização do modelo

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 *chi2* (e.g. *f_classif*, *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).

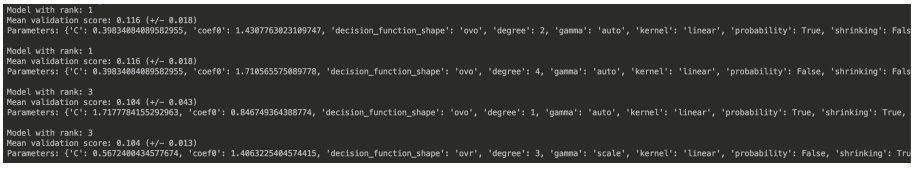


Figura 10: Resultado final da hiper-parametrização do modelo (com *feature selection* manual após estudo das distribuições)

# 7 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.

O resultado obtido é, no entanto, baixo face ao *dataset* que nos foi apresentado. Assim sendo, como trabalho futuro, o grupo sugere que sejam aplicadas otimizações mais avançadas em relação àquelas descritas neste relatório, bem como uma extensão do *dataset* com mais registos ou mais dados sobre informações ainda não contempladas.

# 8 Anexos

## 8.1 Gráficos da distribuição das *features* do *dataset*

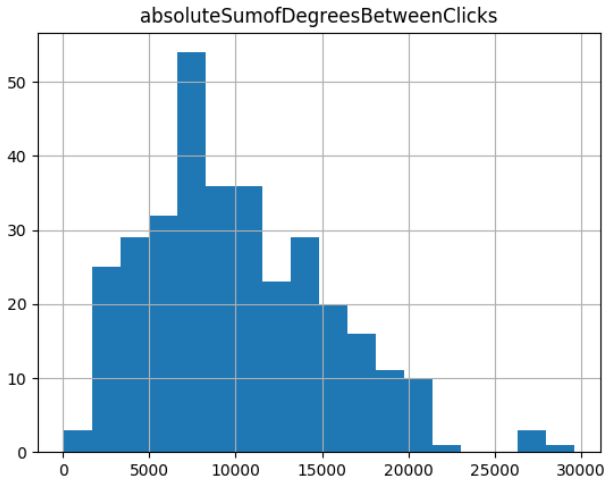


Figura 11: Distribuição de *absoluteSumofDegreesBetweenClicks*

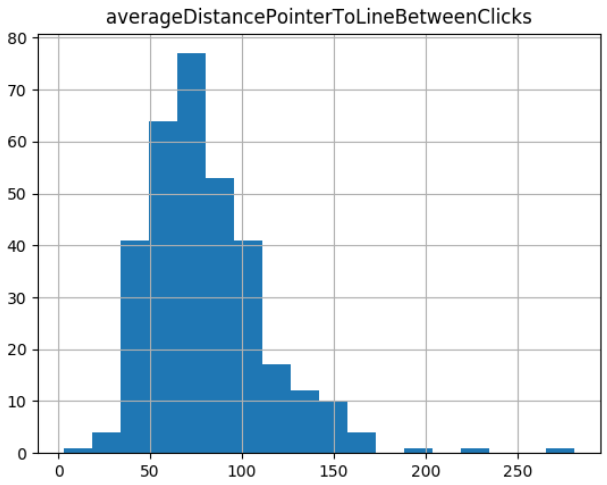


Figura 12: Distribuição de *averageDistancePointerToLineBetweenClicks*

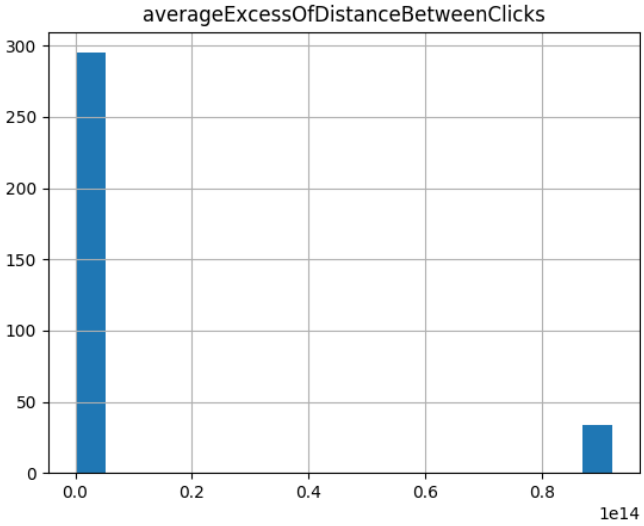


Figura 13: Distribuição de *averageExcessOfDistanceBetweenClicks*

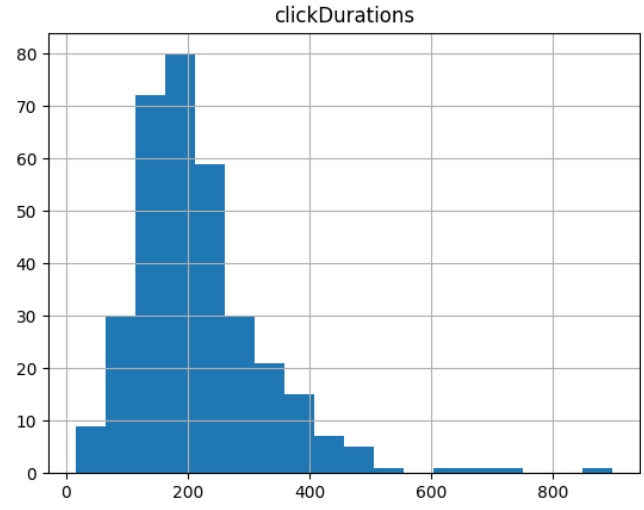


Figura 14: Distribuição de *clickDurations*

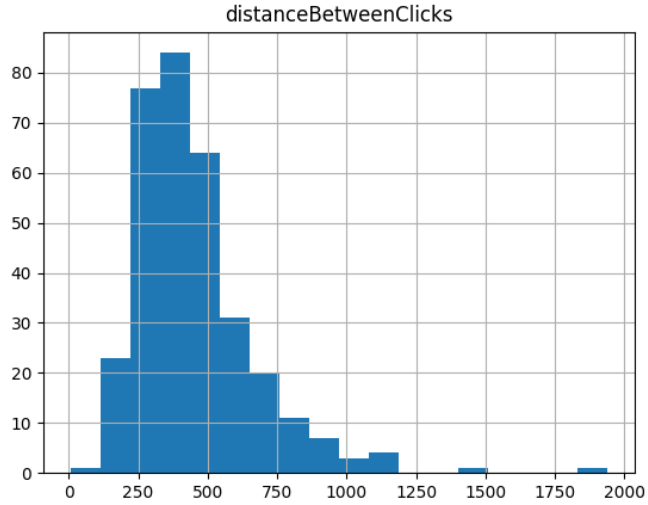


Figura 15: Distribuição de *distanceBetweenClicks*

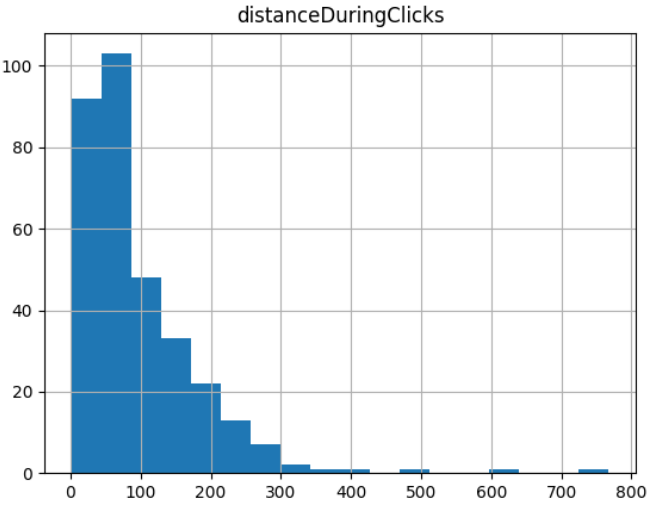


Figura 16: Distribuição de *distanceDuringClicks*

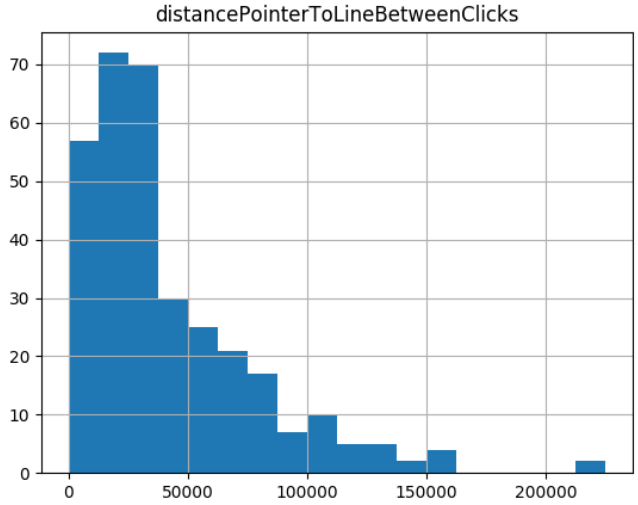


Figura 17: Distribuição de *distancePointerToLineBetweenClicks*

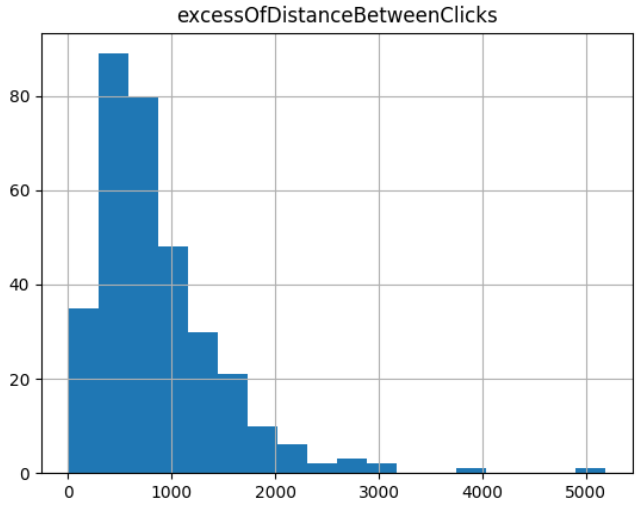


Figura 18: Distribuição de *excessOfDistanceBetweenClicks*

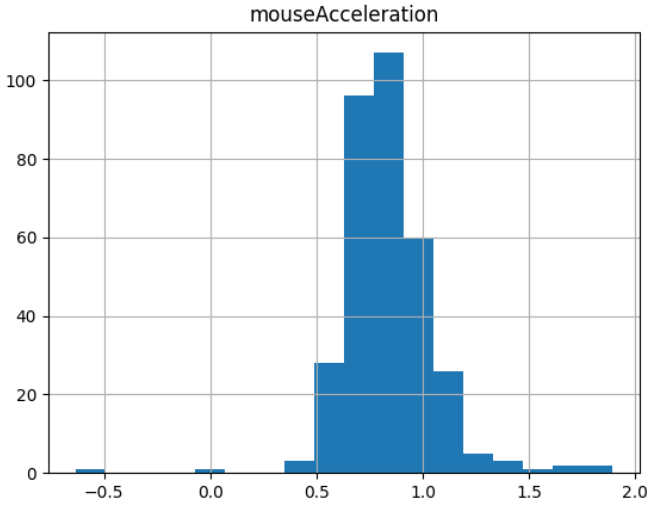


Figura 19: Distribuição de *mouseAcceleration*

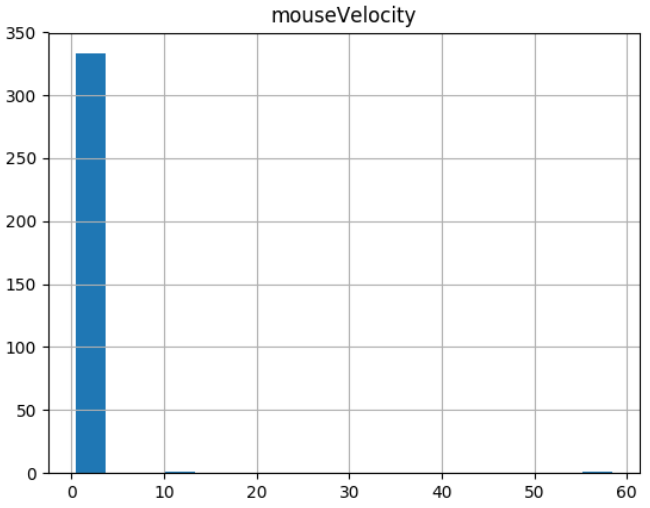


Figura 20: Distribuição de *mouseVelocity*

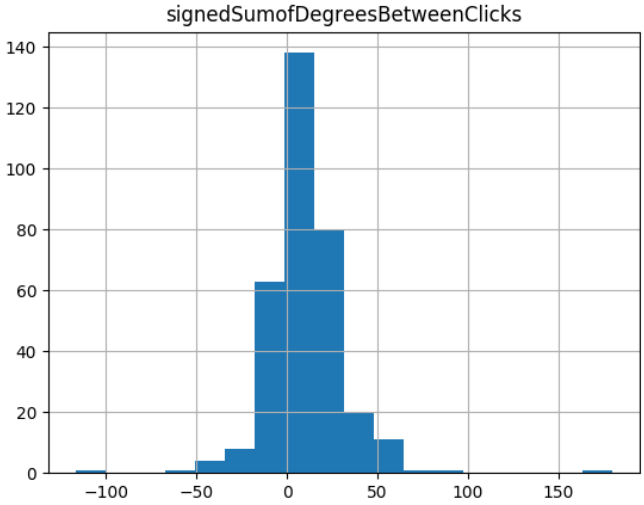


Figura 21: Distribuição de *signedSumofDegreesBetweenClicks*

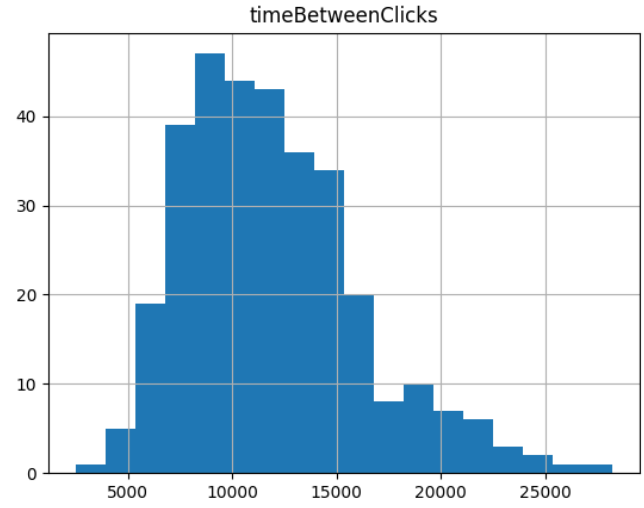


Figura 22: Distribuição de *timeBetweenClicks*

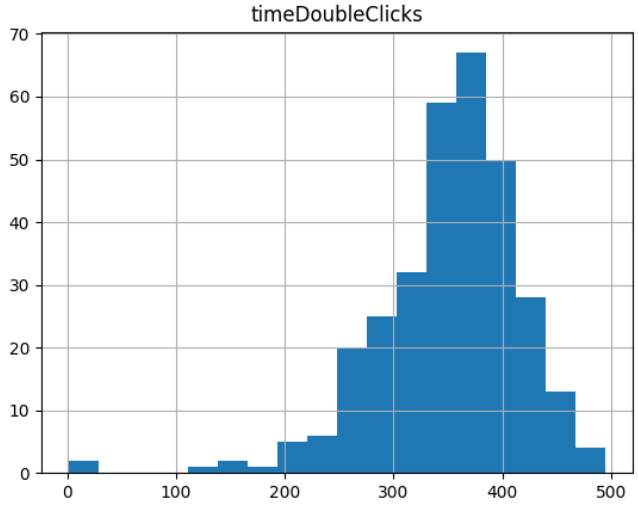


Figura 23: Distribuição de *timeDoubleClicks*