Desafio PicPay

Bruno Paes
github.com/Brunopaes/picpay-sherock_holmes

Sherock_Holmes:
Financial Fraud detector

São Paulo
2020

Sumário

Estrutura do Projeto

Este projeto – hospedado no github (github.com/Brunopaes/picpay-sherock_holmes) – segue a estrutura de diretórios ilustrado na figura 1.

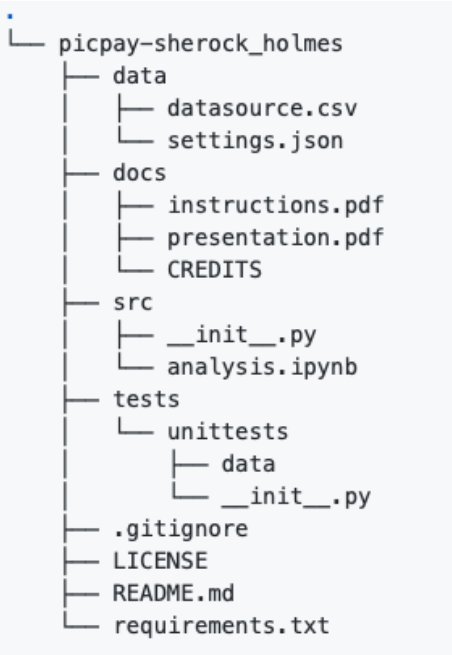

figura 1 – estrutura de diretórios do projeto

Diretórios:

- **data:** diretório para dados e arquivos de suporte (não-*python*);
- **docs:** diretório para documentação;
- **src:** diretório de códigos-fonte;
- **tests:** testes unitários para funções e classes.

O arquivo *requirements.txt* é usado para listar todas as dependências e suas respectivas versões deste projeto.

Análise Exploratória

Esta seção tem por objetivo descrever a metodologia e entender e analisar o comportamento e distribuição dos dados. O comando mostrado abaixo ilustra a forma como o arquivo *.csv foi importado. É importante ressaltar que este arquivo foi baixado, renomeado e movido para o diretório **data** – não havendo alterações manuais no mesmo.

```
In [1]: df = pandas.read_csv('../data/datasource.csv').set_index('Ocorrencia')
```

Utilizando o *pandas*, o arquivo datasource.csv foi importado e a coluna '*Ocorrencia*' foi designada como índice do *dataframe*. Ao final do importe o conteúdo foi designado à variável df. Após esse passo inicial é possível seguir com todas as análises e construções de modelos preditivos.

Os próximos passos eram entender como os dados estavam, de fato, se comportando – média, quartis, valores mínimos e máximos e desvio padrão de cada variável; Se havia a presença de anomalias – *outliers*, valores faltantes ou incongruências nos dados.

O *datarame* é composto por 150.000 linhas. Mas, até então, não se tinha noção do que era uma ocorrência (poderiam ser repetidas? Por quê eram negativas?). Com isso, a primeira ideia foi verificar o comportamento do índice do *datarame* – a variável "Ocorrencia".

```
In [2]:     len(df.index.unique())
```

A linha de comando 2 verifica quantas linhas na variável "Ocorrencia" são únicas no *dataframe*. Caso não houvesse ocorrências duplicadas, o resultado desta linha de comando deveria ser o tamanho total do *dataframe* (150.000 linhas). No entanto, a linha de comando resultou no valor de 64.958 linhas – significando que há valores duplicados.

Com esta simples análise foi possível inferir que uma ocorrência estaria, não somente, relacionada com a transação em si, mas com um cliente/usuário do sistema e, sendo assim, valores repetidos seriam clientes/usuários realizando novas transações.

```
In [3]:     df.describe()
```

Com o entendimento do que seria uma ocorrencia, o próximo passo seria realizar, de fato, a análise exploratória das outras variáveis disponíveis no *dataframe*. Para isso foi-se utilizado a linha de comando 3. Com ela foi possível adquirir algumas métricas úteis para entender o comportamento dos dados de cada variável. O *output* deste comando é apresentado na figura 2.

	PP1	PP2	PP3	PP4	PP5	PP6	PP7	PP8
count	150000.000000	150000.000000	150000.000000	150000.000000	150000.000000	150000.000000	150000.000000	150000.000000
mean	0.058999	-0.000790	-0.192183	-0.037416	0.061588	-0.025715	0.026695	-0.004257
std	1.894453	1.623712	1.406053	1.397615	1.341265	1.310820	1.194923	1.205874
min	-2.454930	-22.057729	-9.382558	-16.875344	-32.911462	-21.307738	-31.527244	-16.635979
25%	-1.243456	-0.802149	-1.138473	-0.812624	-0.526469	-0.424574	-0.527260	-0.340863
50%	0.042647	-0.082193	-0.359076	-0.039549	0.124219	0.245177	-0.013129	-0.037083
75%	0.952018	0.588600	0.555060	0.816575	0.751890	0.734024	0.564334	0.193112
max	36.802320	63.344698	33.680984	5.683171	31.356750	21.929312	43.557242	73.216718

8 rows × 30 columns

figura 2 – análise descritiva do *dataframe*

Com esta tabela – figura 2 – pode-se rapidamente entender como os dados das variáveis estão distribuídos. As medidas de dispersão nos mostram o comportamento dos dados e nota-se que, em sua grande maioria, os dados não muito distantes da média – não sendo maiores que 1,9 desvio-padrões (com exceção da variável “Sacado" que será discutida com mais detalhes). Esse comportamento pode ser observado no *output* da linha de comando 4.

```
In [4]:     df.describe().loc['std'].sort_values(ascending=False).head()
Out [4]:    Sacado    247.302373
            PP1         1.894453
            PP2         1.623712
            PP3         1.406053
            PP4         1.397615
            Name: std, dtype: float64
```

O *output* da linha de comando 4 – que retorna a ordenação decrescente dos desvios padrões – nos mostra que, exceto a variável “Sacado" – com um enorme desvio padrão – as outras variáveis – PP1, PP2 … PP28 – não estão tão dispersas da média. Como mostram as figuras 3 e 4.

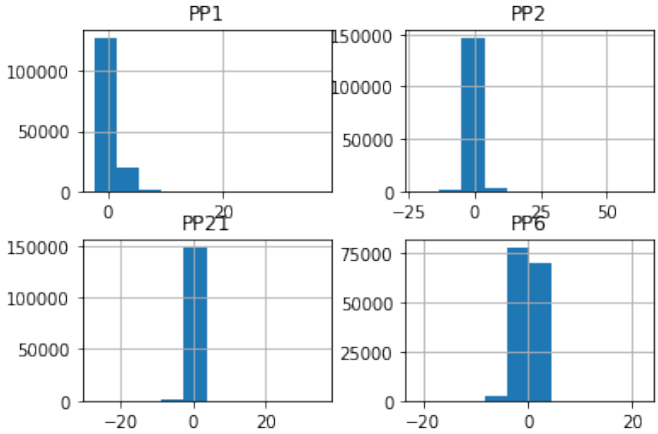

figura 3 – histogramas

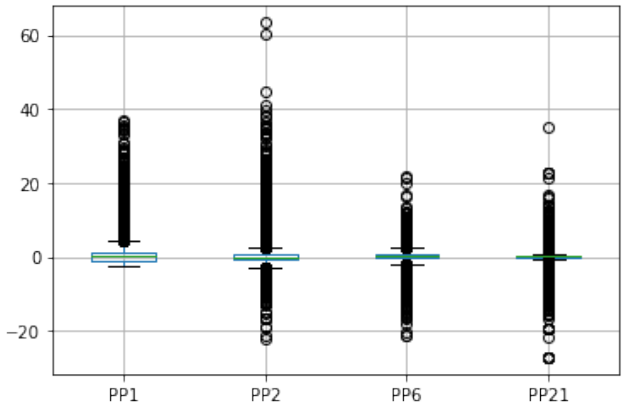

figura 4 – *boxplots*

Como pode-se observar nas figuras 3 e 4. Os dados destas variáveis “aleatoriamente” sortidas – visto que não há como saber o que cada variável representa (e seria inviável fazer o *plot* de 29 variáveis) possuem certa dispersão, mas nada superior à 2 desvios-padrão.

A variável “Sacado"

A variável “Sacado" é a única variável não anonimizada – desconsiderando a ocorrência e a variável *target* (Fraude) – e, por isso e pelo viés que seu *header* trás, uma seção à ela fez-se necessário.

Quando pensa-se em um saque, imagina-se um valor não nulo e positivo – em que retira-se um valor menor ou igual ao seu saldo

disponível. Neste caso em específico, a variável “Sacado" não possui valores positivos e ainda possui valores nulos negativos?. Com isso, inúmeras inferências podem ser realizadas a fim de explicar esse fato – problemas na hora de adquirir os dados, problemas no *parsing* do arquivo original (sql para csv), endereços de memória explodiram e negativaram os valores ou até mesmo uma má interpretação de saque – pode ser que seja um sistema de crédito onde cada valor será retirado da sua conta em um futuro próximo.

```
In [5]:     df[df.Sacado > 0]
```

A linha de comando 5 retorna um *dataframe* vazio – significando que não há valores positivos.

```
In [6]:     df[df.Sacado >= 0]
```

A linha de comando 6 retorna um *dataframe* com 954 linhas. Estas linhas possuem, na variável “Sacado”, o valor de –0.0.

A figura 5 ilustra o quão disperso os dados da variável sacado estão – pode-se notar que num box*plot* contendo outras variáveis, devido à grande dispersão, a escala fica desproporcional.

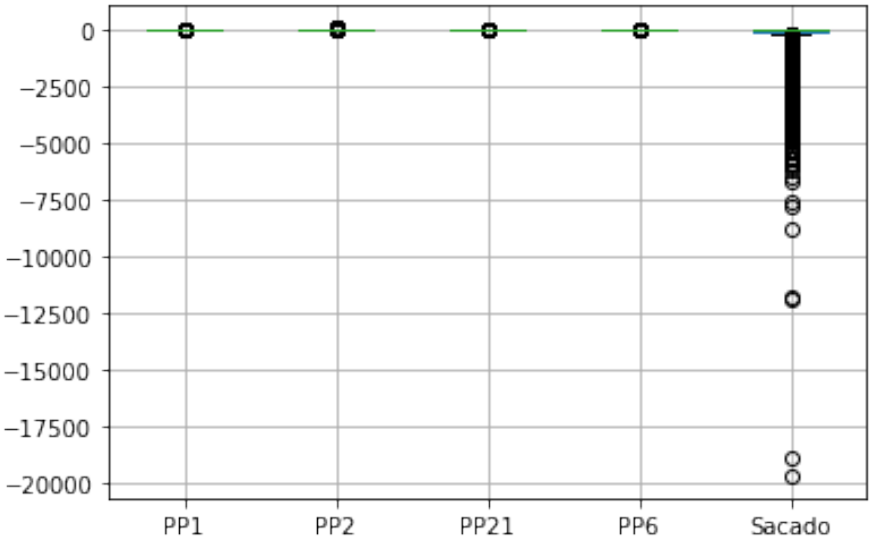

figura 5 – *boxplots* e escala

Buscando por valores N.A.

Algumas anomalias podem ocorrer durante a aquisição de dados. Dentre elas, valores nulos podem ser incômodos durante o pré-processamento. Por conta disso, a biblioteca *pandas* nos possibilita buscar, limpar e, até mesmo, excluir linhas que contenham alguma forma de valores nulos.

```
In [7]:     sum(df.index.isna())
```

A linha de comando 7 busca por valores nulos no índice da *dataframe* – variável ocorrência. A mesma resultou em 0 – o que significa que não há valores nulos no índice.

```
In [8]:     dict_na = {
                  'columns': list(df.columns),
                  'na': []
            }

            for i in range(len(df.columns)):
                  dict_na.get('na').append(sum(df[df.columns[i]].isna()))

pandas.DataFrame(dict_na).set_index('columns')
```

A linha de comando 8 busca por valores nulos no *dataframe* inteiro e resulta numa tabela contendo, como índice, o nome da coluna e como valor a quantidade de valores nulos. A figura 6 ilustra o *output*.

columns	na
PP1	0
PP2	0
PP3	0
PP4	0
PP5	0
PP6	0
PP7	0
PP8	0

figura 6 – *dataframe* de N.A.

Ao somar-se os valores da coluna na – figura 8 – obteve-se 0 – o que significa que não há valores nulos no *dataframe*.

Dataframe desbalanceado?

Quando o problema trata-se de aprendizado de máquina supervisionado – classificação ou regressão – um grande problema, antes mesmo do início do desenvolvimento dos modelos preditivos é o de desbalanceamento de classes. Neste caso, a classificação é binária, ou seja, ou a transação é **Fraudulenta** (1) ou a transação é **Não-Fraudulenta** (0) e portanto, para treinar o modelo, espera-se que 50% das transações no

material de treinamento sejam **Fraude** e 50% sejam **Não-Fraude**. Caso esse cenário não ocorra, o modelo preditivo – RNA, Regressão Linear, SVM, Naive Bayes, Árvores de decisão entre outros – após o treinamento, será enviesado para uma classe ou outra (Pois treinou mais para uma classe do que para outra).

```
In [9]:     df.Fraude.unique()
Out [9]:    array([0, 1])
```

A linha de comando 9 verifica se este cenário é uma classificação binária. Seu *output* nos retorna um *array* contendo 0 e 1 – o que nos indica a classificação binária. A figura 7 indica a proporcionalidade de cada classe no *dataframe*.

Value	Meaning	Total	Percentage
0	Non Fraud	149.763	99,842 %
1	Fraud	237	0,0158 %

figura 7 – Proporcionalidade de classes

Ao se observar a figura 7, nota-se que dos 150.000 registros disponíveis no *dataframe*, apenas 237 ou 0,0158 % do mesmo são transações fraudulentas – indicando um altíssimo nível de desproporcionalidade. Tal desproporcionalidade, durante a construção dos modelos preditivos, acarretará em um possível *overfitting* transações que são realmente fraudulentas.

Redução de dimensionalidade

A redução de dimensionalidade faz-se necessária para otimizar o desempenho dos classificadores. Como o *dataframe* possuía 29 atributos de classificação (PP1, PP2 … PP28, Sacado) era necessário eliminar algumas destas variáveis com o intuito de otimizar, principalmente o desempenho dos classificadores. Para isso um modelo de regressão linear foi criado – que treinou baseado nos 29 atributos – e por meio de correlação e outras técnicas estatísticas definiu quais variáveis independentes (x) mais impactavam na variável dependente (Fraude).

```
In [10]:    lm = linear_model.LinearRegression().fit(x, y)
            attr_reduction = SelectFromModel(lm, prefit=True)
            df_pca = pandas.DataFrame(attr_reduction.transform(x))
```

Outras técnicas mais robustas poderiam ser utilizadas – como, por exemplo, o PCA. Entretanto, optou-se, por conta de sua simplicidade e

desempenho, utilizar a regressão linear para reduzir a dimensionalidade do *dataframe* – que diminuiu de 29 variáveis para somente 9.

MinMax Standardization

Outra questão de pré-processamento, a normalização. Quando as variáveis foram analisadas, pôde-se observar que os valores mínimos e máximos possuíam um alto grau de amplitude. Para otimizar, os atributos de classificação foram normalizados utilizando o MinMax. Esse algoritmo realiza o quociente de todos elementos de cada atributo pelo maior valor de cada atributo – resultando assim em um *dataframe* de baixa amplitude com o valor mínimo sendo 0 e o maior sendo 1.

```
In [11]:    scaler = preprocessing.MinMaxScaler().fit(df_pca)
            df_pca_norm = pandas.DataFrame(scaler.transform(df_pca))
            df_pca_norm['Occurrence'] = occurrence
            df_pca_norm.set_index('Occurrence', drop=True, inplace=True)
```

Data Separation

Ainda no pré-processamento é necessário que o *dataframe* original seja separado em algumas fatias. Algumas bibliografias indicam 80/20 – 80% do material destinado à treinamento do modelo e 20% para testes – outras, por sua vez, indicam 80/10/10 – sendo os últimos 10% destinados à validação – no entanto, independentemente da bibliografia, do autor e da época em que foi escrita, a ideia de se fatiar o *dataframe* tem o mesmo propósito de validar os modelos treinados com dados que o mesmo nunca viu.

Neste caso, o *dataframe* foi dividido em 80/12/8 – 80% para treinamento, 12% para testes e 8% para validação. A figura 8 ilustra a divisão. Vale ressaltar que durante a divisão a proporcionalidade foi mantida.

Sample	Volume	Percentage
Train	120.000	80%
Test	18.000	12%
Validation	12.000	8%

figura 8 – Divisão do *dataframe*

```
In [12]:    def data_separation(df, proportion=0.2):
            """
            Data separation method.
            """
            return train_test_split(df, test_size=proportion)

In [13]:    train, test = data_separation(df)
            test, validation = data_separation(test, 0.4)

In [14]:    # Splitting into train - x and y
            x_train = pandas.DataFrame(train[train.columns[0:-1]])
            y_train = pandas.DataFrame(train[train.columns[-1]])

            # Splitting into test - x and y
            x_test = pandas.DataFrame(test[test.columns[0:-1]])
            y_test = pandas.DataFrame(test[test.columns[-1]])

            # Splitting into validation - x and y
            x_validation =
                  pandas.DataFrame(validation[validation.columns[0:-1]])
            y_validation =
                  pandas.DataFrame(validation[validation.columns[-1]])
```

As linhas de comando 12, 13 e 14 demonstram como os dados foram divididos conforme as regras mencionadas anteriormente.

Benchmarking

Esta seção tem por objetivo descrever o processo de benchmarking de 3 modelos preditivos implementados. Os modelos foram treinados nos dados não normalizados e sem redução de dimensionalidade (O modelo com melhor desempenho nesta etapa será escolhido para *"produção*"). Vale ressaltar que, para evitar viés, todos os modelos treinaram e só depois os resultados foram validados e analisados. Os modelos escolhidos foram:

- Regressão Linear Múltipla;
- Support Vector Machine;
- Random Forest.

```
In [15]:    # Multiple Linear Regression
            begin = datetime.datetime.now()
            lm = linear_model.LinearRegression().fit(x_train, y_train)
            time_screening(begin)

            y_train['Predicted'] = lm.predict(x_train)
            y_train['Predicted'] = y_train['Predicted'].astype(int)

            y_test['Predicted'] = lm.predict(x_test)
            y_test['Predicted'] = y_test['Predicted'].astype(int)

            y_validation['Validation'] = lm.predict(x_validation)
```

```
y_validation['Validation'] =
      y_validation['Validation'].astype(int)
```

A linha de comando 15 ilustra o treinamento e a predição nas amostras de treinamento, testes e validação. A regressão linear, por conta de sua simplicidade, demorou 0:00:00.337315 para convergir. As figuras 9, 10 e 11 ilustram a matriz de confusão para, respectivamente, as amostras de treinamento, testes e validação.

	Non Fraud	Fraud
Non Fraud	119800	1
Fraud	168	31

figura 9 – treinamento

	Non Fraud	Fraud
Non Fraud	17977	0
Fraud	19	4

figura 10 – testes

	Non Fraud	Fraud
Non Fraud	11985	0
Fraud	12	3

figura 11 – validação

Ao observar as matrizes de confusão nota-se que houveram diversos Falsos Negativos em todos os três cenários – caracterizando *underfitting* em transações fraudulentas.

```
In [16]:    # SVM
            begin = datetime.datetime.now()
            lsvm = LinearSVC(C=0.01, penalty="l1", dual=False,
                  max_iter=10000).fit(x_train, y_train.Fraude.values)

            time_screening(begin)

            y_train['Predicted'] = lsvm.predict(x_train)
            y_train['Predicted'] = y_train['Predicted'].astype(int)

            y_test['Predicted'] = lsvm.predict(x_test)
            y_test['Predicted'] = y_test['Predicted'].astype(int)

            y_validation['Validation'] = lsvm.predict(x_validation)
            y_validation['Validation'] =
                  y_validation['Validation'].astype(int)
```

A linha de comando 16 ilustra o treinamento e a predição nas amostras de treinamento, testes e validação. O modelo de SVM demorou 0:00:03.284036 para convergir. As figuras 12, 13 e 14 ilustram a matriz de confusão para, respectivamente, as amostras de treinamento, testes e validação.

	Non Fraud	Fraud
Non Fraud	119788	13
Fraud	54	145

figura 12 – treinamento

	Non Fraud	Fraud
Non Fraud	17974	3
Fraud	8	15

figura 13 – testes

	Non Fraud	Fraud
Non Fraud	11982	3
Fraud	2	13

figura 14 – validação

De forma similar ao que ocorreu com a regressão linear, o modelo SVM também apresentou, embora em menor quantidade, Falsos Negativos em todos os três cenários.

```
In [17]:    # Random Forest
            begin = datetime.datetime.now()
            r_forest =
                  RandomForestClassifier(n_estimators=90).fit(x_train,
            y_train.Fraude.values)

            time_screening(begin)

            y_train['Predicted'] = r_forest.predict(x_train)
            y_train['Predicted'] = y_train['Predicted'].astype(int)

            y_test['Predicted'] = r_forest.predict(x_test)
            y_test['Predicted'] = y_test['Predicted'].astype(int)

            y_validation['Validation'] = r_forest.predict(x_validation)
            y_validation['Validation'] =
                  y_validation['Validation'].astype(int)
```

A linha de comando 17 ilustra o treinamento e a predição nas amostras de treinamento, testes e validação. O modelo de Random Forest, demorou 0:01:50.102289 para convergir – quando comparado ao tempo dos outros dois modelos testados, o random forest possui um péssimo desempenho no quesito tempo polinomial. As figuras 15, 16 e 17 ilustram a matriz de confusão para, respectivamente, as amostras de treinamento, testes e validação.

	Non Fraud	Fraud
Non Fraud	119801	0
Fraud	0	199

figura 15 – treinamento

	Non Fraud	Fraud
Non Fraud	17976	1
Fraud	7	16

figura 16 – testes

	Non Fraud	Fraud
Non Fraud	11982	3
Fraud	2	13

figura 17 – validação

De forma similar ao que ocorreu com a regressão linear e com o modelo SVM, o modelo random forest também apresentou, embora em menor quantidade que a regressão linear e o SVM, Falsos Negativos nos cenários de teste e validação – caracterizando *overfitting*. A figura 18 sumariza os resultados e demonstra as taxas de assertividade de cada modelo.

	linear_model	svm	random_forest
train	0.998592	0.999442	1.000000
test	0.998944	0.999389	0.999556
validation	0.999000	0.999583	0.999583

figura 18 – sumarização das taxas de assertividade

A regressão linear, embora simples e rápida, foi o modelo com a pior assertividade. As elevadas taxas de assertividade apresentada na figura 18 escondem os elevados casos de Falsos Negativos – transações fraudulentas que são classificadas como não fraudes (pior cenário de erro).

A SVM obteve um melhor desempenho quando comparada à regressão linear – até obteve um tempo de treinamento aceitável (polinomial quando comparado ao random forest) – entretanto, ao contrário da regressão linear, a SVM obteve diversos casos de Falsos Positivos – transações não fraudulentas que são classificadas como fraudes.

O random forest obteve um excelente desempenho no quesito assertividade – parece, dentre os três, ter o menor *overfitting* e não apresenta *underfitting* – entretanto, seu tempo de execução foi, ao menos, 5 vezes maior que os outros dois modelos.

Em suma, considerando não somente a assertividade do modelo e número de casos em que houve uma classificação errônea – sendo Falsos Negativos menos tolerantes – mas considerando também o seu tempo de treinamento e convergência, o modelo escolhido para ir para *"produção"* foi o random forest – mesmo embora tenha convergido em tempo não polinomial.

Classificador

Após a última etapa, o modelo *"campeão"* foi o random forest. A ideia nesta etapa é usar este modelo em um *dataframe* otimizado e reduzido. Portanto, a linha de código 17 foi novamente utilizada – mas agora foi usada com os dados normalizados pelo *MinMax* e reduzidos pela regressão linear. Vale ressaltar que as otimizações feitas no dataframe acarretaram numa melhora significativa do tempo de treinamento – que foi de 0:01:50.102289 para 0:00:48.581284 (Um decréscimo de 0:01:01.521005).

As figuras 19, 20 e 21 ilustram a matriz de confusão para, respectivamente, as amostras de treinamento, testes e validação.

	Non Fraud	Fraud
Non Fraud	119815	0
Fraud	0	185

figura 19 – treinamento

	Non Fraud	Fraud
Non Fraud	17968	2
Fraud	8	22

figura 20 – testes

	Non Fraud	Fraud
Non Fraud	11977	1
Fraud	7	15

figura 21 – validação

As matrizes de confusão do random forest nessa etapa, quando comparada as matrizes de confusão da etapa anterior, são muito similares – ou seja, as mudanças de dimensionalidade e normalização do *dataframe* não influenciaram o modelo preditivo negativamente.

Entretanto, mesmo com um excelente desempenho, a preocupação do modelo estar apresentando *overfitting* em casos de fraude ainda não foi confirmada ou refutada. Para testar esta hipótese, os *dataframes* de testes e validação normalizados e reduzidos irão se fundir e, para este novo teste, somente transações fraudulentas serão selecionadas e passadas ao modelo preditor. *Overfitting* será caracterizado caso o modelo apresente uma taxa de assertividade baixa.

```
In [18]:   # Checking if there's overfitting on classifying Frauds – due the
           low quantity of data entries
           overfitting = x_validation
           overfitting['Fraude'] = y_validation['Fraude']

           aux = x_test
           aux['Fraude'] = y_test['Fraude']

           overfitting = overfitting.append(aux)
           overfitting = overfitting[overfitting['Fraude'] == 1]
           del(aux)

In [19]:   overfitting['Predicted'] =
                 r_forest.predict(overfitting.drop(columns=['Fraude']))



In [20]:   print(len(overfitting[overfitting['Fraude'] ==
                 overfitting['Predicted']])/len(overfitting))
Out [20]:  0.7115384615384616

In [21]:   pandas.DataFrame(confusion_matrix(
           overfitting[['Fraude']], overfitting[['Predicted']]),
           ['Non Fraud', 'Fraud'], ['Non Fraud', 'Fraud'])
```

Conclusão

As linhas de comando 18, 19, 20 e 21 demonstram o processo no qual os *dataframes* de testes de validação foram fundidos em um único, o *select* de somente transações fraudulentas, passando estes dados para o preditor e a construção da matriz de confusão e a impressão da taxa de assertividade. Nota-se que o *output* da linha de comando 20 é uma taxa de assertividade muito inferior as taxas obtidas anteriormente. A figura 22 ilustra a matriz de confusão deste teste.

	Non Fraud	Fraud
Non Fraud	0	0
Fraud	15	37

figura 22 – matriz de confusão – *overfitting*

Por fim, a matriz de confusão confirma a hipótese de *overfitting*. Sendo assim, com 29% dos casos sendo Falsos Positivos – transações fraudulentas que são consideradas não fraudes – esse modelo não pode ir para produção (diferentemente do *underfitting* que uma solução iminente é utilizar técnicas de grid search ou modelos mais robusto), pois para o treinamento do modelo é necessário que haja mais dados de transações fraudulentas.

Vale ressaltar que, neste cenário, Falsos Negativos são o grande gargalo do classificador e que, até o momento, esse classificador é muito bom em classificar transações não fraudulentas. Para se corrigir esse gargalo, como já mencionado, é necessário que novos dados de transações fraudulentas – gerados artificialmente ou não – sejam adicionados à amostra de treinamento.

Referências Bibliográficas

Normalization with Scikit-learn. Acesso em 27 de dezembro de 2019. Disponível em: <https://towardsdatascience.com/scale-standardize-or-normalize-with-scikit-learn-6ccc7d176a02>

Function's documentation – pandas and sklearn.