Projeto1

June 26, 2021

1 Projeto 1 - SCC0270 - Neural Networks and Deep Learning

1.1 Primeiro Projeto Prático

1.1.1 Nome: William Luis Alves Ferreira

1.1.2 NºUSP: 9847599

1.2 Instalação de pacotes pip

[1]:
```
#biblioteca para salvar a sessão do jupiter
!pip install dill
```

```
Requirement already satisfied: dill in c:\users\will_\anaconda3\lib\site-
packages (0.3.4)
```

[2]:
```
import dill

dill.load_session('notebook_env.db')
```

[3]:
```
!pip install tensorflow
```

[4]:
```
!pip install keras
```

1.3 Carregamento de bibliotecas base

[5]:
```
#Manipulação de vetores e matrizes
import numpy as np
import pandas as pd
#Gerencimento de plotagem
import matplotlib.pyplot as plt

#Arquitetura de modelos para as redes neurais (Keras e Tensorflow)
from keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import roc_auc_score, make_scorer, confusion_matrix,␣
↪plot_confusion_matrix
```

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import GridSearchCV
from sklearn import metrics
from sklearn.linear_model import SGDClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
```

1.4 Carregamento dos dados fornecidos

[6]:
```
data_train = pd.read_csv("./train.csv",delimiter=',')
```

2 Questão 1

Definições para questão 1

2.0.1 Definição de acuracia

Temos como acuracia a taxa de acertos da predição por toda a amostra, ou seja, consideramos alto o indice quando acertos (verdadeiros positivos e verdadeiros negativos) são a maioria da ocorrencia da predição das classes.

$$Acuracia = \frac{VP + VN}{VP + VN + FP + FN}$$

2.0.2 Definindo AUC

ROC - *receiver operating characteristic curve*, traduzido como **curva característica de operação do receptor**

AUC - *Area Under the ROC Curve* , traduzido como **Área Sob a Curva ROC**

No qual plotamos uma cruva (ROC) de **taxa de verdadeiro positivo** (TVP, sigla em inglês, TPR) vs. **taxa de falso positivo** (TFP, sigla em inglês, FPR) e pode ser calculada como

$$\text{Sensibilidade (Recall)} = TVP = \frac{VP}{P} = \frac{VP}{VP + FN}$$

$$\text{Erro} = TFP = \frac{FP}{N} = \frac{FP}{FP + VN}$$

A AUC é invariante em estala, pois não trata dos valores absolutos predicionados, ao inves disso, trata da relação de acertos e erros, além dos verdadeiros e falsos possitivos. Uma interpletação possível é tendo o beneficio do modelo como **TVP** e maleficio como **TFP**, o ponto ideal seria na relação de maior o TVP e menor o TFP, ou seja, com a relação mais próxima do ideal temos a área sob a curva próxima a 1 caso contrário próximo a 0.

2.0.3 a) Baseado na base de dados fornecida, qual das duas métricas de avaliação deve ser usada para medir os resultados dos modelos: Acurácia ou AUC?

2.0.4 b) Na base fornecida, qual seria o resultado esperado de um modelo aleatório para cada uma das duas métricas (Acurácia e AUC)?

Primeiro realizaremos um estudo sobre os dados e averiguar sua distribuição e tendências, afim de identificar desvios que possam influenciar as métricas de avaliação do modelo.

Verificando os dados de teste

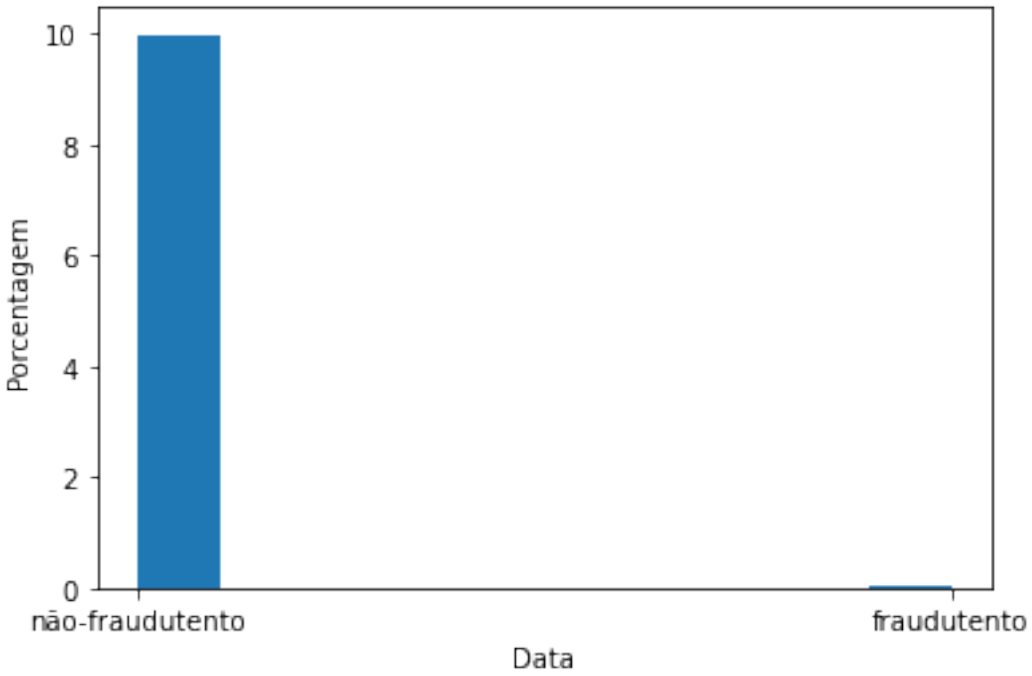
[8]:
```
# classes substituindo valores para facilitar plot
classes = data_train['Class']

classes[classes==0]="não-fraudutento"
classes[classes==1]="fraudutento"
```

```
<ipython-input-8-a8564a62bd7f>:4: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  classes[classes==0]="não-fraudutento"
```

[9]:
```
plt.hist(classes, density=1)
plt.ylabel('Porcentagem')
plt.xlabel('Data');
```



[10]:
```
#quantidade de transações não fraudulentas
total = data_train['Class'].describe()[0]
q_nf = data_train['Class'].describe()[3]
#quantidade de transações fraudulentas
q_f = data_train['Class'].describe()[0] - data_train['Class'].describe()[3]

data_train['Class'].describe()
```

[10]:
```
count              142404
unique                  2
top       não-fraudutento
freq               142135
Name: Class, dtype: object
```

[11]:
```
print("Porcentagem das classes no dataset de treino")
print("Não Fraudulenta ", (q_nf/total)*100)
print("Fraudulenta ", (q_f/total)*100)
```

```
Porcentagem das classes no dataset de treino
Não Fraudulenta  99.81110081177495
Fraudulenta  0.18889918822504986
```

No cenário de flaudes temos um desbalanceamento entre as classes alvo (**não-fraudulentas** e **fraudulenta**), pois esperace que a classe majoritaria em transações seja **não-fraudulenta**, logo para esse cenário existe o risco de inviezar nosso modelo para ignorar as fraudes, tonando-se, assim, otimista em relação a análise de um verdadeiro positivo (transação que realmente é não fraudulenta), o que não atende a necessidade desse tipo de problema, já que o foca é encontrar falsos positivos, ou seja, transações tidas como não-fraudulentas que na realidade são fraudulentas. Como exemplo dado em aula, se um modelo é desenvolvido para classificar transações **não-fraudulentas** e **fraudulenta** e espera-se que 95% das transações seja não-fraudulenta, nossa acurácia poderia chegar em 95%, porem não atenderia a necessidade de conferi a classe **não-fraudulenta** com precisão, logo precisamos verificar tanto a taxa de verdadeiro positivo e falso positivo, assim contemplando o objetivo de indentificar fraudes.

Portanto, como não podemos balancear a base de testes, precisamos adequar a métrica de avaliação do modelo para expressar tanto a taxa de verdadeiro positivo e verdadeiro negativo, neste sentido podemos utilizar a **AUC** para avaliarmos de forma precisa o problema.

A comparação entre as métricas de avaliação do modelo será comparada na questão 2.

2.1 Questão 2

Nesta questão temos o interesse de investigar os impactos de valores do parâmetro de regularização L2

Com as variações de L2:

[0.0, 0.01, 0.1, 1, 10]

[12]:
```
data_train = pd.read_csv("./train.csv",delimiter=',')
data_test = pd.read_csv("./test.csv",delimiter=',')
```

[13]:
```
target_test = data_test.drop('Class', axis=1)
label_test = data_test['Class']
```

[14]:
```
target_train = data_train.drop('Class', axis=1)
label_train = data_train['Class']
```

[15]:
```
label_test.shape, label_train.shape
```

[15]:
```
((142403,), (142404,))
```

Para o *label* de teste entrar no modelo, temos que reformular a classe única para duas classes binárias, através de To_categorical()

[16]:
```
label_train = to_categorical(label_train, 2)
label_test  = to_categorical(label_test, 2)
```

[17]:
```
label_test
```

[17]:
```
array([[1., 0.],
       [0., 1.],
       [1., 0.],
       ...,
       [1., 0.],
       [1., 0.],
       [1., 0.]], dtype=float32)
```

Prosseguindo, vamos gerar 5 modelos de MLP para explorar as variações do regularizador do modelo.

[18]:
```
alphas = [0,0.01,0.1,1,10]
```

[19]:
```
models = []

for a in alphas:
    model = None
    model = MLPClassifier(
        hidden_layer_sizes=[20,20,20,20,20],
        activation='relu',
        alpha=a,
        verbose=True,
        random_state=42
    )
    models.append(model)
```

[20]:
```
for a in range(len(alphas)):
    models[a].fit(target_train, label_train)
```

Como discutido na questão 1 vemos a avaliação dos modelos em ambas as métricas

Acurácia

[21]:
```
scores = []

for a in range(len(alphas)):
    scores.append(models[a].score(target_test, label_test))
```

[22]:
```
scores
```

[22]:
```
[0.9952248197018321,
 0.9978441465418566,
 0.998939629080848,
 0.9993469238709859,
 0.9984340217551596]
```

E, verificamos que a taxa da métrica de acurácia retrata aproximadamente a **taxa de não-fraude**, pois não consegue retratar o carater de excessão dos dados desbalanciados entre as classes fruto da natureza do problema

[23]:
```
scores = []

for a in range(len(alphas)):
    scores.append(roc_auc_score(label_test, models[a].predict(target_test)))
```

AUC

[24]:
```
scores
```

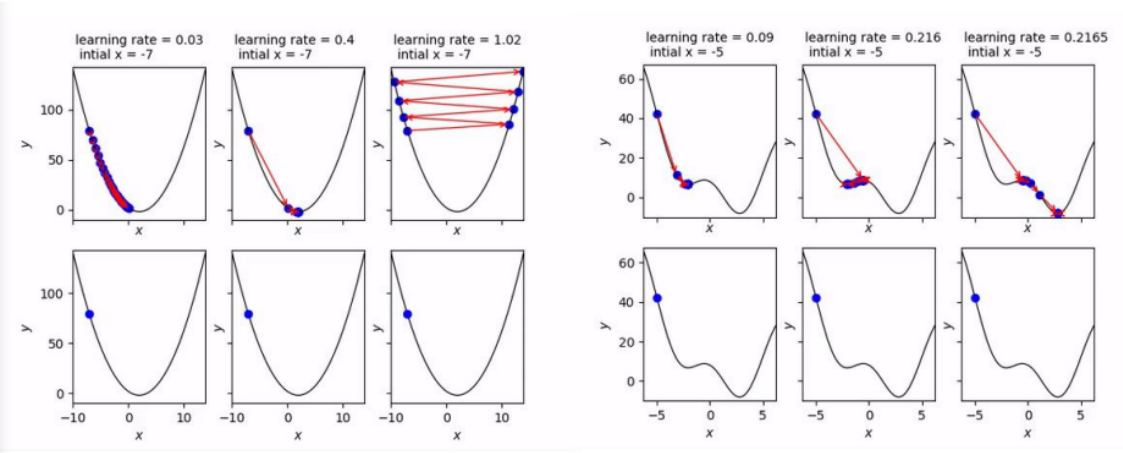
[24]:
```
[0.8610712231132518,
 0.8724392183974461,
 0.9009690078956315,
 0.7981957280198724,
 0.5]
```

Verificamos que o terceiro modelo correspondente ao regulador L2 = 0.1, e esta relacionado ao termo α relacionado a taxa de aprendizado exposto pelas expressões:

$$w = w - \alpha \frac{\partial E}{\partial w}$$

$$b = b - \alpha \frac{\partial E}{\partial b}$$

Resumidamente, temos que a derida do erro em função dos parâmetros nos oferece a **Direção** (tangente) da função, ou seja, indica se devemos diminuir ou aumentar o valor do parâmetro com forme a inclinação da reta tangente. Já o parâmetro α com o passo das atualização dos parâmetros *bias* e *weight*, indica o quanto deste valor deve ser ajustado, desta forma, devemos atentar-se para não iginorar o ponto de minimo numa função de erro, como ilustra a figura a direita.



Fonte: Material de aula do docente Tiago Santana de Nazaré

Portanto, para nosso modelo o regulador L2 – α escolhido é 0.1

2.2 Questão 3

Nesta questão vamos realizar a análise dos hiperparâmetros e arquitetura dos modelos *Multilayer Perceptron* (MLP) e
K-Nearest Keighboors (KNN) em termos dos parametros, respectivamente, **hidden_layer_sizes** e **n_neighbors**.

Para desenvolver a avaliação do modelo será desenvolvido um script para avaliação dos K-Folds, ou seja, avaliar as três variações de cada modelo com a base de treino dividindo-a em 3 partes, e em seguida utiliza-se a classe **GridSearchCV** que automatiza o processo do script desenvolvido.

[25]:
```
data_train = pd.read_csv("./train.csv",delimiter=',')
data_test = pd.read_csv("./test.csv",delimiter=',')
```

[26]:
```
target_test = data_test.drop('Class', axis=1)
label_test = data_test['Class']
```

[27]:
```
target_train = data_train.drop('Class', axis=1)
label_train = data_train['Class']
```

Para facilitar a distinção declaro cada modelo de forma explicita a seguir

[28]:
```
redes = []

model = MLPClassifier(
    hidden_layer_sizes=(),
    verbose=True
```

```
)
redes.append(model)
model = MLPClassifier(
    hidden_layer_sizes=[10],
    verbose=True
)
redes.append(model)
model = MLPClassifier(
    hidden_layer_sizes=[5,5],
    verbose=True
)
redes.append(model)
```

[29]:
```
knns = []
knn = KNeighborsClassifier(n_neighbors=3)
knns.append(knn)
knn = KNeighborsClassifier(n_neighbors=5)
knns.append(knn)
knn = KNeighborsClassifier(n_neighbors=7)
knns.append(knn)
```

2.2.1 Preparando o k-fold, onde k=3

[30]:
```
kf = StratifiedKFold(n_splits=3)
```

[31]:
```
kf.get_n_splits(data_train)
```

[31]: 3

[32]:
```
#   kfold_indices[grupo][train/test], train = 0 , test = 1

kfold_indices = []
for train_index, test_index in kf.split(data_train,label_train):
    print("TRAIN:", train_index.shape, "TEST:", test_index.shape)
    kfold_indices.append([train_index,test_index])
```

```
TRAIN: (94936,) TEST: (47468,)
TRAIN: (94936,) TEST: (47468,)
TRAIN: (94936,) TEST: (47468,)
```

Como resultado, temos um array composto por 3 grupos e composto por *fetures* e *labes* da base de teste e traino, como ilustrado:

data_kfold[grupo][train_target | train_label | test_target | test_label]

[33]:
```
#  data_kfold[grupo][train_target|train_label|test_target|test_label]
#  train_target = 0, train_label = 1 , test_target= 2, test_label=3
data_kfold = []

for i in range(len(kfold_indices)):          #groups
        data_kfold.append([target_train.iloc[kfold_indices[i][0]],
                           label_train.iloc[kfold_indices[i][0]],
                           target_train.iloc[kfold_indices[i][1]],
                           label_train.iloc[kfold_indices[i][1]]])
```

[34]:
```
data_kfold
```

2.3 Comparando hiperparâmetros

Com o treinamento de cada modelo (3) com cada um dos k-fold (3), resultando 9 variações dos modelos, e tirado média aritimética entre os 3 *scores* de cada variação de parâmetro dos modelos.

2.3.1 Treinamento dos modelos de MLP

[35]:
```
%%time
models_k = []

cv_k = []
for m in range(len(redes)):
    for cv in range(kf.get_n_splits(data_train)):
        cv_k.append(redes[m].fit(data_kfold[cv][0], data_kfold[cv][1]))
    models_k.append(cv_k)
    cv_k = []
```

[36]:
```
models_k
```

[36]:
```
[[MLPClassifier(hidden_layer_sizes=(), verbose=True),
  MLPClassifier(hidden_layer_sizes=(), verbose=True),
  MLPClassifier(hidden_layer_sizes=(), verbose=True)],
 [MLPClassifier(hidden_layer_sizes=[10], verbose=True),
  MLPClassifier(hidden_layer_sizes=[10], verbose=True),
  MLPClassifier(hidden_layer_sizes=[10], verbose=True)],
 [MLPClassifier(hidden_layer_sizes=[5, 5], verbose=True),
  MLPClassifier(hidden_layer_sizes=[5, 5], verbose=True),
  MLPClassifier(hidden_layer_sizes=[5, 5], verbose=True)]]
```

A seguir, realiza-se a avaliação das 9 variações do modelo com a métrica AUC.

[37]:
```
scores_k = []

socres_cv_k = []
for m in range(len(redes)):
```

```
    for cv in range(kf.get_n_splits(data_train)):
        socres_cv_k.append(roc_auc_score(data_kfold[cv][3], redes[m].
↪predict(data_kfold[cv][2])))
    scores_k.append(socres_cv_k)
    socres_cv_k = []
```

[38]:
```
scores_k = np.array(scores_k)
select_M = 0
Tmax = 0
param_MLP = ['()', '[10]', '[5,5]']
print("--- Avaliação dos modelos ---")
print("Média")
print("Desvio Padrão")
for i in range(3):
    print("--- Modelo: hidden_layer_sizes", param_MLP[i])
    if(np.mean(scores_k[i])>Tmax):
        select_M = i
        Tmax = np.mean(scores_k[i])
    print(np.mean(scores_k[i]))
    print(np.std(scores_k[i]))
```

```
--- Avaliação dos modelos ---
Média
Desvio Padrão
--- Modelo: hidden_layer_sizes ()
0.730574418681378
0.06953386705355805
--- Modelo: hidden_layer_sizes [10]
0.8811828440327928
0.04378613276874017
--- Modelo: hidden_layer_sizes [5,5]
0.9015734235067513
0.03885168548643127
```

Melhor modelo MLP

[39]:
```
redes[select_M]
```

[39]:
```
MLPClassifier(hidden_layer_sizes=[5, 5], verbose=True)
```

2.3.2 Treinando modelos knn

[40]:
```
knn_k = []

cv_k = []
for m in range(len(knns)):
    for cv in range(kf.get_n_splits(data_train)):
```

```
        cv_k.append(knns[m].fit(data_kfold[cv][0], data_kfold[cv][1]))
    knn_k.append(cv_k)
    cv_k = []
```

[41]:
```
knn_k
```

[41]:
```
[[KNeighborsClassifier(n_neighbors=3),
  KNeighborsClassifier(n_neighbors=3),
  KNeighborsClassifier(n_neighbors=3)],
 [KNeighborsClassifier(), KNeighborsClassifier(), KNeighborsClassifier()],
 [KNeighborsClassifier(n_neighbors=7),
  KNeighborsClassifier(n_neighbors=7),
  KNeighborsClassifier(n_neighbors=7)]]
```

[42]:
```
%%time
scores_knn_k = []

scores_cv_k = []
for m in range(len(redes)):
    for cv in range(kf.get_n_splits(data_train)):
        scores_cv_k.append(roc_auc_score(data_kfold[cv][3], knns[m].
 ↪predict(data_kfold[cv][2])))
    scores_knn_k.append(scores_cv_k)
    scores_cv_k = []
```

```
Wall time: 10min 53s
```

[43]:
```
scores_knn_k
```

[43]:
```
[[0.9550245201792572, 0.9055450021341364, 0.7832805662262372],
 [0.9437674587257284, 0.9055450021341364, 0.7888361217817927],
 [0.943756905527054, 0.905523895291298, 0.7943811239159292]]
```

[44]:
```
select_K= 0
Tmax = 0
param_KNN = ['3', '5', '7']
print("--- Avaliação dos modelos ---")
print("Média")
print("Desvio Padrão")
for i in range(3):
    print("--- Modelo: n_neighbors", param_KNN[i])
    if(np.mean(scores_knn_k[i])>Tmax):
        select_K = i
        Tmax = np.mean(scores_knn_k[i])
    print(np.mean(scores_knn_k[i]))
    print(np.std(scores_knn_k[i]))
```

```
--- Avaliação dos modelos ---
Média
Desvio Padrão
--- Modelo: n_neighbors 3
0.8812833628465436
0.07218248534349225
--- Modelo: n_neighbors 5
0.8793828608805526
0.06590029346438546
--- Modelo: n_neighbors 7
0.8812206415780937
0.06335754171772501
```

Melhor modelo Knn

[45]:
```
knns[select_K]
```

[45]:
```
KNeighborsClassifier(n_neighbors=3)
```

2.4 Utilizando a classe GridSearchCV, temos

2.4.1 MLP explorando hiperparâmetros

[46]:
```
parameters = [{'hidden_layer_sizes':[(), [10], [5,5]]}]
```

[47]:
```
cv_mlp = GridSearchCV(estimator=MLPClassifier(), param_grid=parameters, cv=3,␣
 ↪scoring=make_scorer(roc_auc_score, greater_is_better=True))
```

[48]:
```
%%time
cv_mlp.fit(target_train, label_train)
```

```
Wall time: 1min 12s
```

[48]:
```
GridSearchCV(cv=3, estimator=MLPClassifier(),
             param_grid=[{'hidden_layer_sizes': [(), [10], [5, 5]]}],
             scoring=make_scorer(roc_auc_score))
```

[49]:
```
cv_mlp.cv_results_
```

[49]:
```
{'mean_fit_time': array([5.06447442, 6.90484198, 8.26344283]),
 'std_fit_time': array([0.34937208, 1.241977  , 1.06794019]),
 'mean_score_time': array([0.01041857, 0.02329834, 0.02117562]),
 'std_score_time': array([0.00736704, 0.00638249, 0.00714339]),
 'param_hidden_layer_sizes': masked_array(data=[(), list([10]), list([5, 5])],
              mask=[False, False, False],
        fill_value='?',
             dtype=object),
 'params': [{'hidden_layer_sizes': ()},
```

```
 {'hidden_layer_sizes': [10]},
 {'hidden_layer_sizes': [5, 5]}],
'split0_test_score': array([0.83661028, 0.8924986 , 0.91466814]),
'split1_test_score': array([0.79443389, 0.83883612, 0.82220112]),
'split2_test_score': array([0.62775667, 0.79993668, 0.83323835]),
'mean_test_score': array([0.75293361, 0.84375713, 0.85670254]),
'std_test_score': array([0.09017266, 0.03794812, 0.0412348 ]),
'rank_test_score': array([3, 2, 1])}
```

[53]:
```
cv_mlp.best_params_
```

[53]:
```
{'hidden_layer_sizes': [5, 5]}
```

2.4.2 KNN explorando hiperparâmetros

[54]:
```
parameters = [{'n_neighbors':[3, 5, 7]}]
```

[55]:
```
cv_knn = GridSearchCV(estimator=KNeighborsClassifier(), param_grid=parameters,␣
↪cv=3, scoring=make_scorer(roc_auc_score, greater_is_better=True), verbose=1 ,␣
↪n_jobs=-1)
```

[56]:
```
%%time
cv_knn.fit(target_train, label_train)
```

```
Fitting 3 folds for each of 3 candidates, totalling 9 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   4 out of   9 | elapsed:  2.8min remaining:  3.5min
[Parallel(n_jobs=-1)]: Done   9 out of   9 | elapsed:  4.4min finished

Wall time: 4min 23s
```

[56]:
```
GridSearchCV(cv=3, estimator=KNeighborsClassifier(), n_jobs=-1,
             param_grid=[{'n_neighbors': [3, 5, 7]}],
             scoring=make_scorer(roc_auc_score), verbose=1)
```

[57]:
```
cv_knn.cv_results_
```

[57]:
```
{'mean_fit_time': array([2.64343643, 2.59269675, 2.98557075]),
 'std_fit_time': array([0.07750696, 0.21274195, 0.23690055]),
 'mean_score_time': array([162.79603028, 184.54077792, 182.89834396]),
 'std_score_time': array([46.76201868, 42.20994226, 52.31914229]),
 'param_n_neighbors': masked_array(data=[3, 5, 7],
              mask=[False, False, False],
        fill_value='?',
             dtype=object),
 'params': [{'n_neighbors': 3}, {'n_neighbors': 5}, {'n_neighbors': 7}],
 'split0_test_score': array([0.95461295, 0.95463405, 0.95463405]),
```

```
 'split1_test_score': array([0.78330167, 0.76662445, 0.75551334]),
 'split2_test_score': array([0.78328057, 0.78883612, 0.79438112]),
 'mean_test_score': array([0.84039839, 0.83669821, 0.83484284]),
 'std_test_score': array([0.08076188, 0.08388479, 0.0861786 ]),
 'rank_test_score': array([1, 2, 3])}
```

[58]:
```
cv_knn.best_params_
```

[58]:
```
{'n_neighbors': 3}
```

2.5 Comparação script próprio e o GridSearchCV

2.5.1 MLP

[59]:
```
print("'hidden_layer_sizes':",redes[select_M].get_params()['hidden_layer_sizes'])
```

```
'hidden_layer_sizes': [5, 5]
```

[60]:
```
cv_mlp.best_params_
```

[60]:
```
{'hidden_layer_sizes': [5, 5]}
```

2.5.2 KNN

[61]:
```
knns[select_K]
```

[61]:
```
KNeighborsClassifier(n_neighbors=3)
```

[62]:
```
cv_knn.best_params_
```

[62]:
```
{'n_neighbors': 3}
```

2.6 Experiência de performance

2.6.1 a) Qual técnica obteve o melhor resultado na base de teste?

2.6.2 b) Qual técnica demora mais para gerar as previsões nos dados de teste?

2.6.3 Melhor modelo MLP

[63]:
```
model = redes[select_M]
```

TEMPO DE EXECUÇÃO

TREINO:

[64]:
```
 %%time
model.fit(target_train, label_train)
```

```
Iteration 1, loss = 0.06482553
Iteration 2, loss = 0.00929426
Iteration 3, loss = 0.00674939
Iteration 4, loss = 0.00546009
Iteration 5, loss = 0.00462534
Iteration 6, loss = 0.00409635
Iteration 7, loss = 0.00389245
Iteration 8, loss = 0.00372930
Iteration 9, loss = 0.00356336
Iteration 10, loss = 0.00345263
Iteration 11, loss = 0.00330815
Iteration 12, loss = 0.00320408
Iteration 13, loss = 0.00312258
Iteration 14, loss = 0.00303033
Iteration 15, loss = 0.00297433
Iteration 16, loss = 0.00289251
Iteration 17, loss = 0.00286500
Iteration 18, loss = 0.00280397
Iteration 19, loss = 0.00276909
Iteration 20, loss = 0.00273327
Iteration 21, loss = 0.00271190
Iteration 22, loss = 0.00265369
Iteration 23, loss = 0.00265341
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs.
Stopping.
Wall time: 11.5 s
```

[64]:
```
MLPClassifier(hidden_layer_sizes=[5, 5], verbose=True)
```

PREDIÇÃO BASE:

[65]:
```
%%time
predict_model = model.predict(target_test)
```

```
Wall time: 52 ms
```

TEMPO TOTAL:

Aproximadamente: 12 s

[66]:
```
print("Score AUC - MLP:",roc_auc_score(label_test, predict_model))
```

```
Score AUC - MLP: 0.9064892478239231
```

2.6.4 Melhor modelo KNN

[67]:
```
knn = knns[select_K]
```

TEMPO DE EXECUÇÃO

TREINO:

[68]:
```
%%time
knn.fit(target_train, label_train)
```

```
Wall time: 1.69 s
```

[68]:
```
KNeighborsClassifier(n_neighbors=3)
```

PREDIÇÃO BASE:

[69]:
```
%%time
predict_knn = knn.predict(target_test)
```

```
Wall time: 12min 24s
```

TEMPO TOTAL:

Aproximadamente 12 min 30 s

[70]:
```
print("Score AUC - MLP:",roc_auc_score(label_test, predict_knn))
```

```
Score AUC - MLP: 0.8740350449471302
```

Portanto, o melhor modelo entre MLP e KNN para a solução deste problema vista tempo de execução total e avaliação é **MLP**.

Observação: Houve dificuldade em obter o output do comando mágico %%time, por esse motivo considerar os valores totais de tempo em uma execução durante a experiÊncia e pode não se repetir em re-execuções, porém a disparidade entre KNN e MLP é constante.

2.7 Questão 4

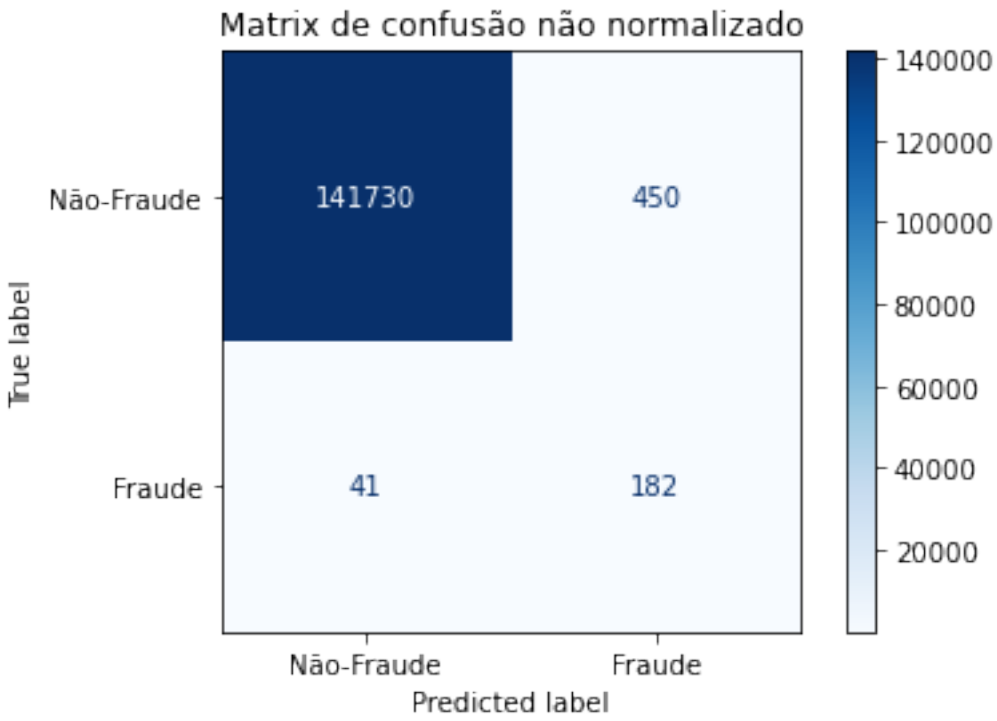
Para esta questão avaliaremos a matrix de confução para encontrar os seguintes indices:

Verdadeiro - Negativo: Cada fraude evitada em média evita um prejuízo (gera um lucro) de R$ 100

Falso - Negativo: Cada não-fraude bloqueada gera em média um prejuízo de 2 reais.

[71]:
```
cm = plot_confusion_matrix(model, target_test, label_test,
                           display_labels=["Não-Fraude", "Fraude"],
                           cmap=plt.cm.Blues,
                           normalize=None)
cm.ax_.set_title("Matrix de confusão não normalizado")

plt.show()
```



[72]:
```
VN = cm.confusion_matrix[1][1]
```

[73]:
```
FN = cm.confusion_matrix[0][1]
```

[74]:
```
Lucro = (VN*(100))+(FN*(-2))
```

Por fim, temos o cálculo do lucro oferecido pelo modelo com os valores expostos no enunciado.

2.7.1 Lucro esperado

[75]:
```
print("R$",Lucro)
```

```
R$ 17300
```

2.8 Questão 5

Nesta questão utilizaremos um modelo composto por um único neurônio para verificar o impacto das *fetures* da base de dados.

[76]:
```
data_train = pd.read_csv("./train.csv",delimiter=',')
data_test = pd.read_csv("./test.csv",delimiter=',')
```

[77]:
```
target_test = data_test.drop('Class', axis=1)
label_test = data_test['Class']
```

[78]:
```
target_train = data_train.drop('Class', axis=1)
label_train = data_train['Class']
```

Inicio o modelo com o hiperparametro **penalty**='l1', pois resulta em *weight* nulos para as entradas sem influência na saída

[79]:
```
neuronio = SGDClassifier(loss='log', random_state=42, penalty='l1')
```

[80]:
```
neuronio.fit(target_train, label_train)
```

[80]:
```
SGDClassifier(loss='log', penalty='l1', random_state=42)
```

[81]:
```
roc_auc_score(label_test, neuronio.predict(target_test))
```

[81]:
```
0.8295832447595324
```

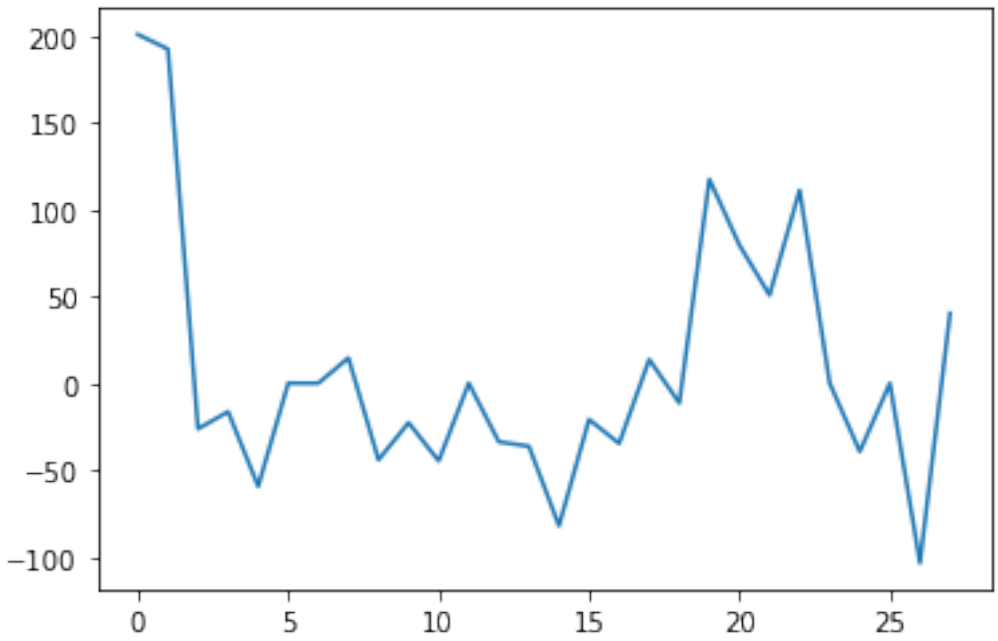
[82]:
```
w = neuronio.coef_[0]
```

[83]:
```
w
```

[83]:
```
array([ 200.65772195,  192.20966766,  -26.254422  ,  -16.38961516,
        -59.29673962,    0.        ,    0.        ,   14.55805129,
        -44.07702743,  -22.69659509,  -44.70670146,    0.        ,
        -33.73913207,  -36.25067654,  -81.97062719,  -20.86478838,
        -34.66528511,   13.66088106,  -11.35961667,  117.29494357,
         79.43164706,   50.68633693,  111.07959735,    0.        ,
        -39.31362634,    0.        , -103.35595862,   40.21647171])
```

[84]:
```
plt.plot(w)
```

[84]:
```
[<matplotlib.lines.Line2D at 0x20a11f1a610>]
```



Agora, verificamos o impacto das variáveis com peso nulo, primeiro removendo-as da base de dados

[85]:
```
tol = 1
remover = []

for i  in range(len(w)):
    if(np.sqrt(w[i]**2)>tol):
        remover.append(i)
```

[86]:
```
fetures = np.array(data_train.columns)
fetures = np.delete(fetures, remover)
fetures
```

[86]:
```
array(['V6', 'V7', 'V12', 'V24', 'V26', 'Class'], dtype=object)
```

Com a nova base de dados verificamos o modelo e seus pesos

[87]:
```
new_target_train = data_train.drop(fetures, axis=1)
label_train = data_train['Class']
```

[88]:
```
new_target_test = data_test.drop(fetures, axis=1)
label_test = data_test['Class']
```

Em carater de experimento faz-se a comparação da avaliação e matrix de confusão do modelo, em especial, verificando os valores **Verdadeiro Negativo** e **Falso Negativo** que condiz com as taxas intríncicas ao problema.

2.8.1 Comparando modelos de único neurônio

[89]:
```
new_neuronio = SGDClassifier(loss='log', random_state=42, penalty='l1')
```

[90]:
```
new_neuronio.fit(new_target_train, label_train)
```

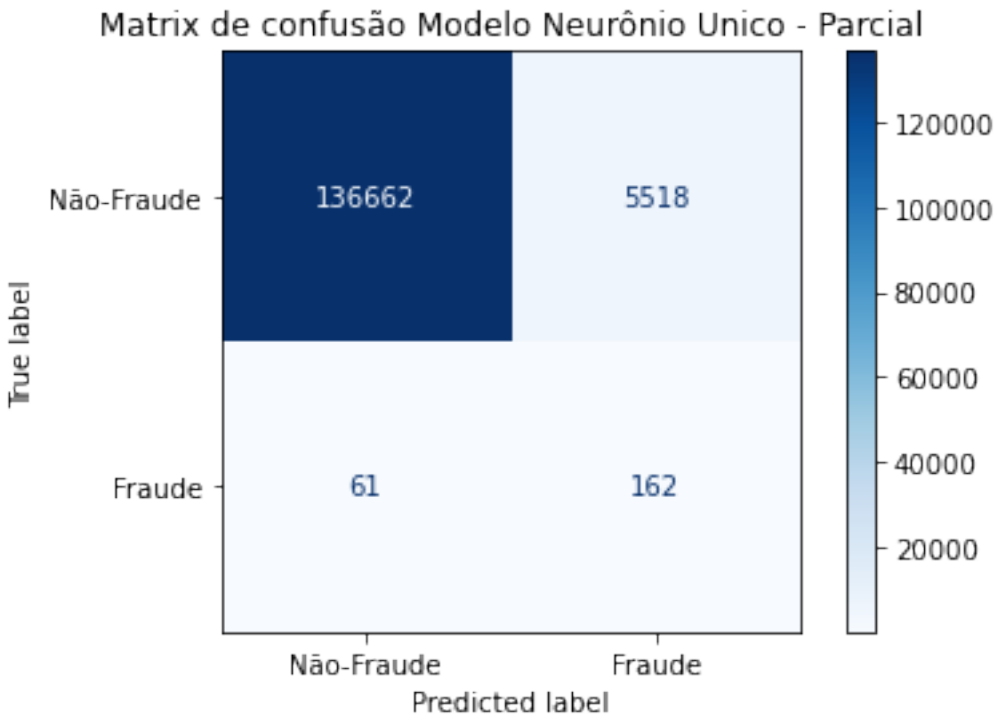
[90]:
```
SGDClassifier(loss='log', penalty='l1', random_state=42)
```

[91]:
```
new_w = new_neuronio.coef_
new_w
```

[91]:
```
array([[ 141.82196849,  126.61731964,  -19.92491524,    0.        ,
         -16.34105026,   21.97532216,  -57.03951522,  -39.62393377,
         -32.42537554,  -40.06476677,  -18.54386709,  -67.46675212,
         -22.98794608,  -25.38343468,   22.74837859,   -2.71104658,
          80.63222529,   91.56267741,   47.45075295,   82.04395061,
         -63.59311157, -115.68936126,    9.3229018 ]])
```

[92]:
```
new_neuronio_cm = plot_confusion_matrix(new_neuronio, new_target_test,␣
 ↪label_test,
                                display_labels=["Não-Fraude", "Fraude"],
                                cmap=plt.cm.Blues,
                                normalize=None)
new_neuronio_cm.ax_.set_title("Matrix de confusão Modelo Neurônio Unico -␣
 ↪Parcial")

plt.show()
```

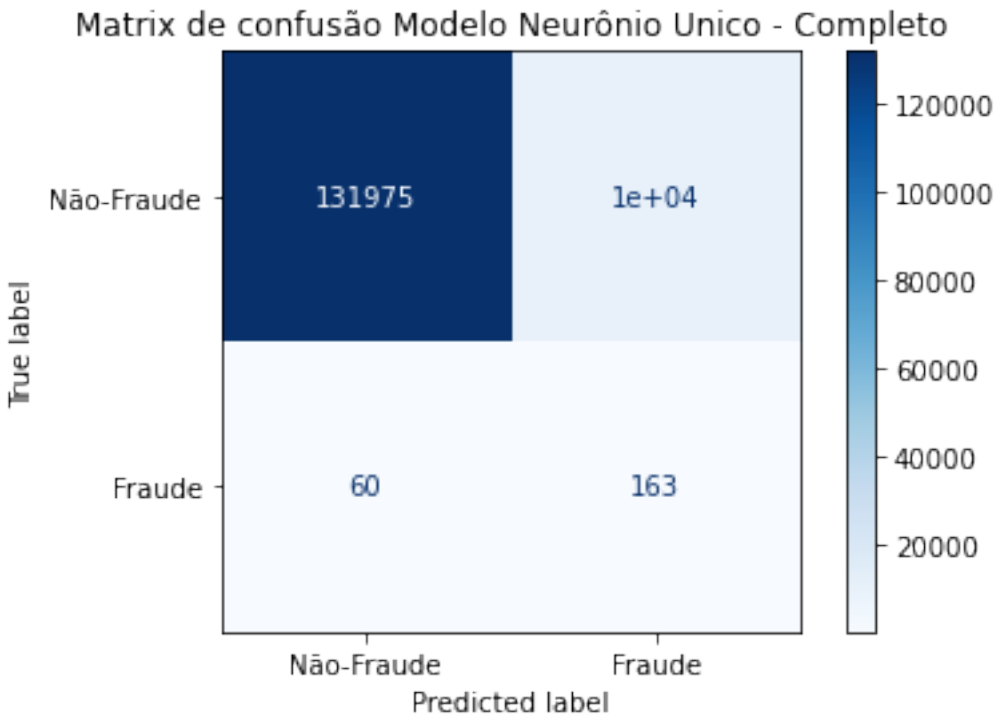


[93]:
```
auc_neuronio_parcial = roc_auc_score(label_test, new_neuronio.
 ↪predict(new_target_test))
auc_neuronio_parcial
```

[93]:
```
0.8438237199482497
```

[94]:
```
neuronio_cm = plot_confusion_matrix(neuronio, target_test, label_test,
                                    display_labels=["Não-Fraude", "Fraude"],
                                    cmap=plt.cm.Blues,
                                    normalize=None)
neuronio_cm.ax_.set_title("Matrix de confusão Modelo Neurônio Unico - Completo")

plt.show()
```



[95]:
```
auc_neuronio_completo = roc_auc_score(label_test, neuronio.predict(target_test))
auc_neuronio_completo
```

[95]: 0.8295832447595324

Diferencia da métrica de avaliação AUC:

[96]:
```
auc_neuronio_completo - auc_neuronio_parcial
```

[96]: -0.014240475188717294

2.8.2 Comparando modelos da questão 3 e 4

[97]:
```
new_model  = MLPClassifier(
    hidden_layer_sizes=redes[select_M].get_params()['hidden_layer_sizes'],
    verbose=True
)
new_model.fit(new_target_train, label_train)
```

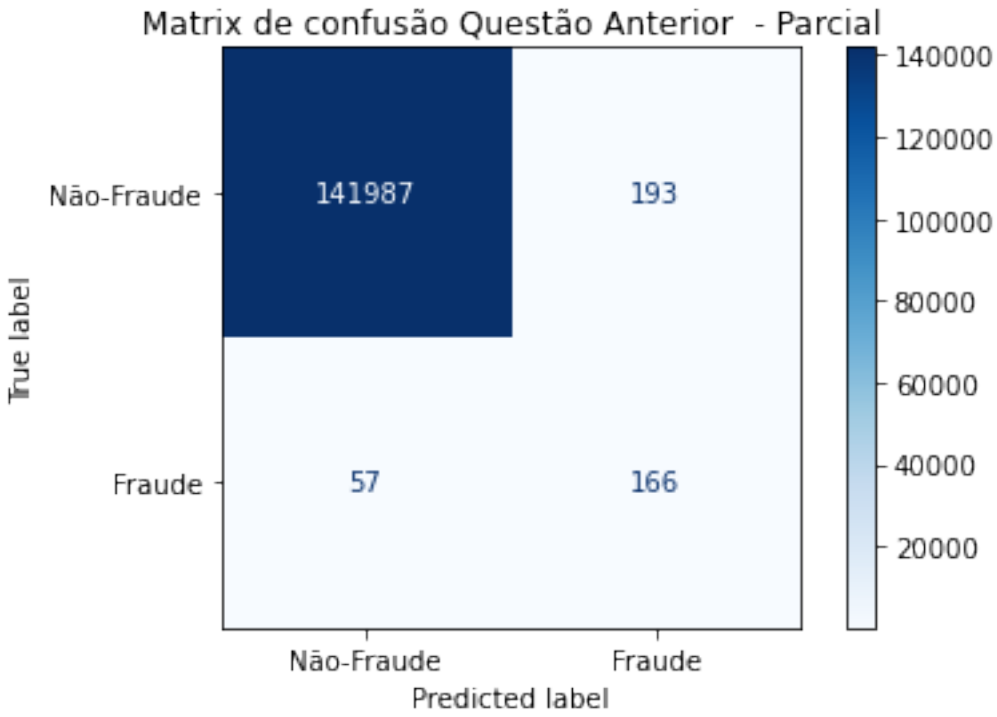
```
Iteration 1, loss = 0.19016008
Iteration 2, loss = 0.00837503
Iteration 3, loss = 0.00622355
Iteration 4, loss = 0.00560224
Iteration 5, loss = 0.00513658
```

```
Iteration 6, loss = 0.00477084
Iteration 7, loss = 0.00447922
Iteration 8, loss = 0.00422771
Iteration 9, loss = 0.00400797
Iteration 10, loss = 0.00386998
Iteration 11, loss = 0.00374902
Iteration 12, loss = 0.00364177
Iteration 13, loss = 0.00354283
Iteration 14, loss = 0.00344591
Iteration 15, loss = 0.00334567
Iteration 16, loss = 0.00328476
Iteration 17, loss = 0.00322285
Iteration 18, loss = 0.00318337
Iteration 19, loss = 0.00314440
Iteration 20, loss = 0.00312816
Iteration 21, loss = 0.00310673
Iteration 22, loss = 0.00308196
Iteration 23, loss = 0.00306930
Iteration 24, loss = 0.00302371
Iteration 25, loss = 0.00303744
Iteration 26, loss = 0.00299506
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs.
Stopping.
```

[97]:
```
MLPClassifier(hidden_layer_sizes=[5, 5], verbose=True)
```

[104]:
```
new_cm = plot_confusion_matrix(new_model, new_target_test, label_test,
                               display_labels=["Não-Fraude", "Fraude"],
                               cmap=plt.cm.Blues,
                               normalize=None)
new_cm.ax_.set_title("Matrix de confusão Questão Anterior  - Parcial")

plt.show()
```

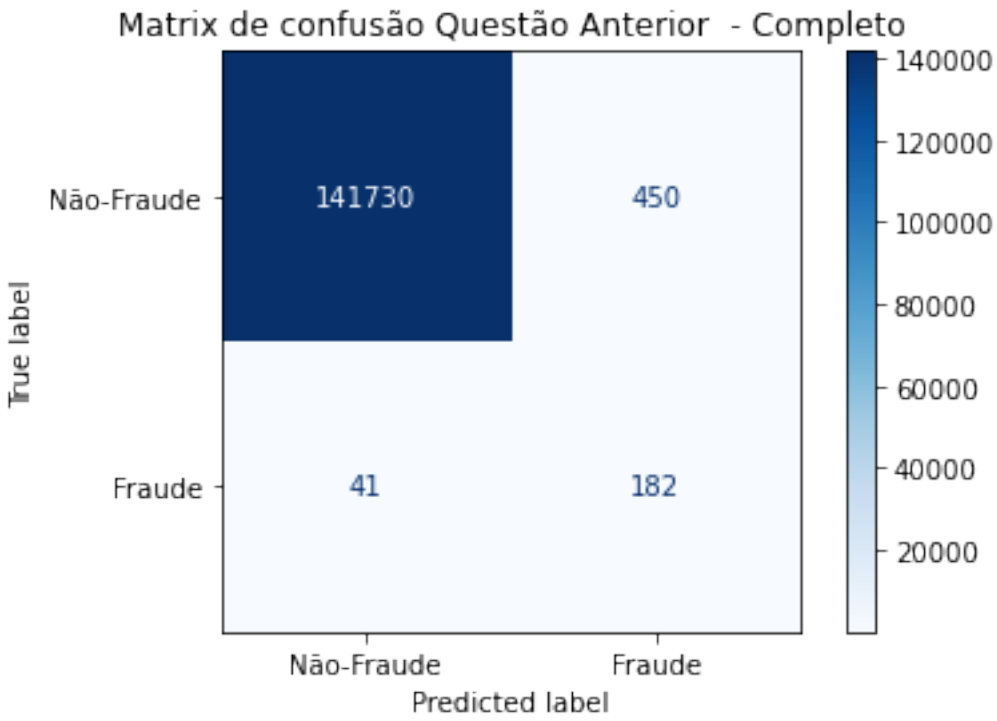


[99]:
```
auc_parcial = roc_auc_score(label_test, new_model.predict(new_target_test))
auc_parcial
```

[99]: 0.8715185922978955

[100]:
```
cm = plot_confusion_matrix(model, target_test, label_test,
                           display_labels=["Não-Fraude", "Fraude"],
                           cmap=plt.cm.Blues,
                           normalize=None)
cm.ax_.set_title("Matrix de confusão Questão Anterior  - Completo")

plt.show()
```



[101]:
```
auc_completo = roc_auc_score(label_test, model.predict(target_test))
auc_completo
```

[101]: 0.9064892478239231

Verificamos através do regulador L1 que as variaveis de menor impacto para o modelo foram:

[102]:
```
fetures = np.delete(fetures, -1)
fetures
```

[102]: array(['V6', 'V7', 'V12', 'V24', 'V26'], dtype=object)

Em relação a avaliação dos modelos Completo e parcial (sem as variaveis com w=0), vemos uma diferença de

[103]:
```
auc_completo - auc_parcial
```

[103]: 0.03497065552602763

que apesar de baixa, não sustenta a ausência das variaveis, pois podem retratar alguma característica que não é retratada na amostra da base de teste. De todo modo foram identificadas as variáveis de menos "importantes" para defenir uma saída no modelo.