Titanic - Aprendizado com o desastre

Contexto

O mais famoso desastre marítimo da história está relatado em tabelas de dados contendo diversas informações sobre os passageiros, inclusive quem sobreviveu ou não ao desastre.

Conteúdo

O seguinte dicionário de dados pode ser usado para melhor entendimento dos atributos.

Variável	Definição	Chave
survival	Sobrevivente (classe)	0 = Não, 1 = Sim
pclass	Classe do bilhete	1 = 1a, 2 = 2a, 3 = 3a
sex	Sexo	

|Age| Idade em anos |sibsp| # irmãos ou companheiros no Titanic | |parch| # pais ou filhos no Titanic | |ticket| Número do bilhete |fare| Tarifa cobrada | | |cabin| Número da cabine | | |embarked| Porto de embarque| C = Cherbourg, Q = Queenstown, S = Southampton|

In [ ]:
```
import pandas as pd
import numpy  as np
import seaborn as sea
import tensorflow as ts
import chart_studio.plotly as py
import plotly.figure_factory as ff
import matplotlib.pyplot as plt
```

```
2023-07-04 20:35:58.923348: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find cuda drivers on your machine, GPU will not b
e used.
2023-07-04 20:35:58.965388: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could not find cuda drivers on your machine, GPU will not b
e used.
2023-07-04 20:35:58.966098: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use avai
lable CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
2023-07-04 20:35:59.823503: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not find TensorRT
```

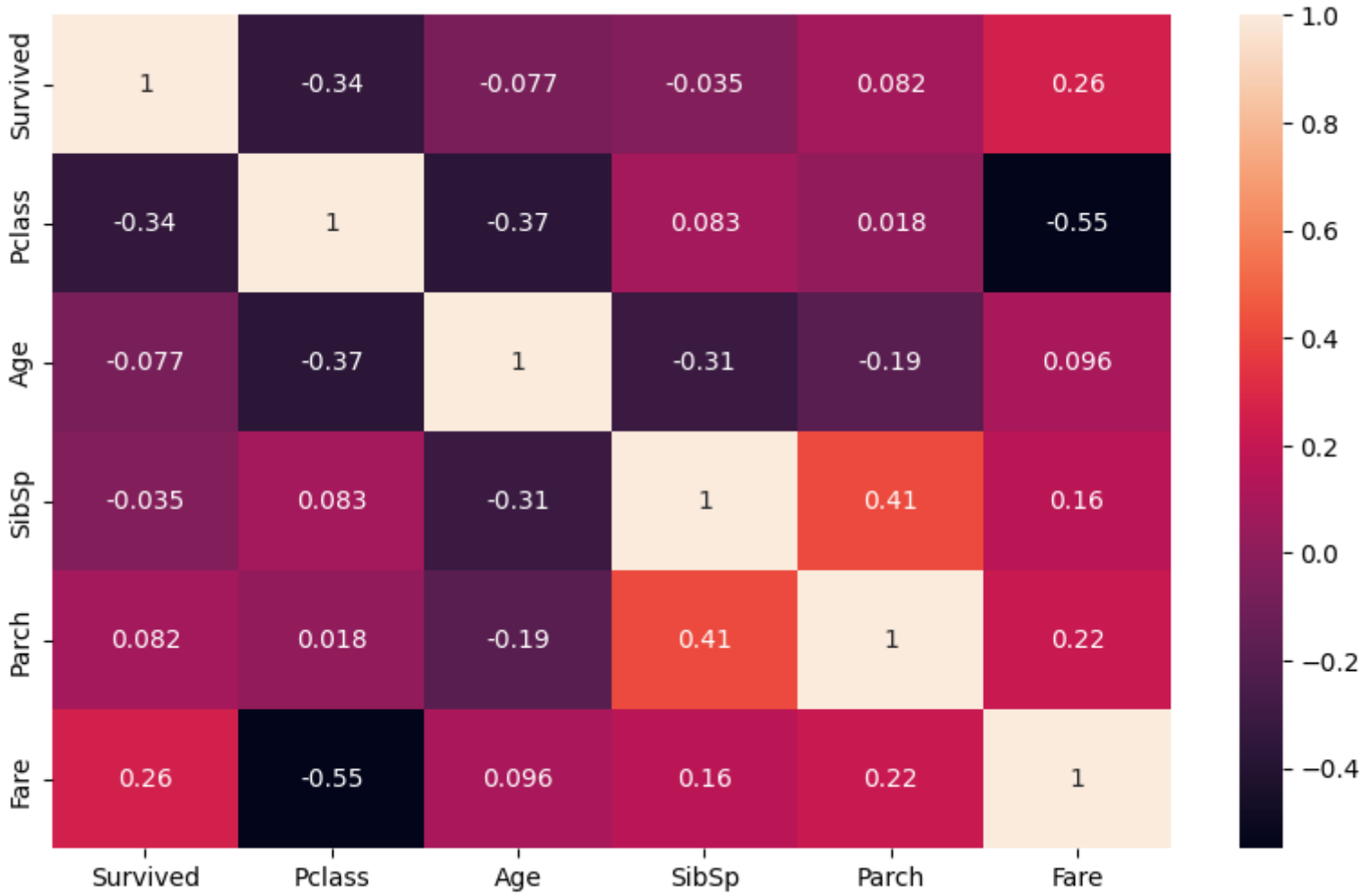
In [ ]:
```
df = pd.read_csv(r'dados/titanic.csv',sep=',')
```

Grafico de correlação de atributos usando o seaborn

- Removi certos atributos que não me dariam uma correlação alta , e também alguns já mostra que não tem nenhuma como `PassegnerID`
  - `PassegnerID` Primary key : atributo que não repetição

In [ ]:
```
plt.figure(figsize=(10,6))
sea.heatmap((df.drop(columns=['PassengerId','Name','Ticket','Cabin','Embarked','Sex'])).corr(),annot=True)
```

Out[ ]: <Axes: >



Usando a biblioteca plotly para mostrar um tabela com da descrição dos dados

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

Out[ ]:

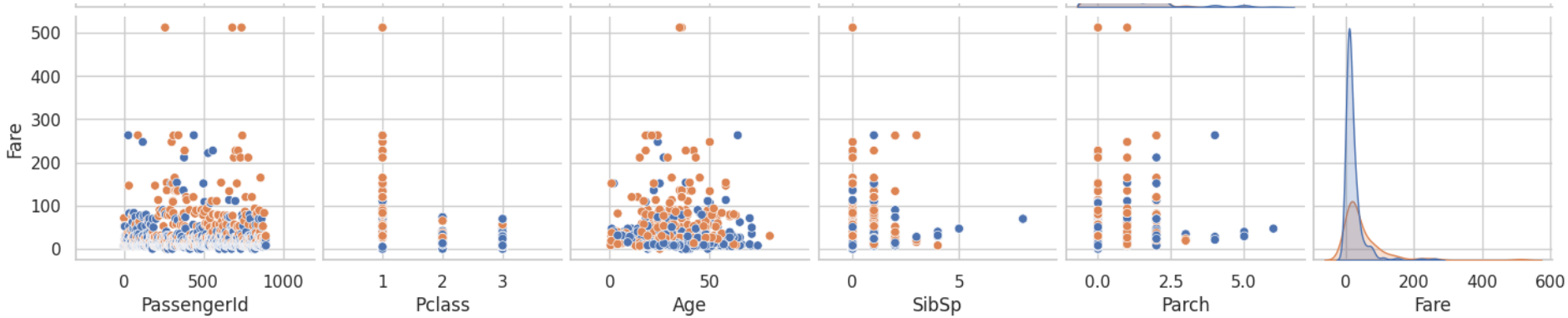
	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

Usei pairplot para ver relacao|associcao| em relação a `Survived`

In [ ]:
```
sea.set_theme(style="whitegrid")
sea.pairplot(df,hue='Survived')
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7f445f739290>
```


PassengerId
Pclass
Age
SibSp
Parch
Survived
0
1



Tratamento de dados

Irei converter para valores binario `Embarked e Pclass`

- Usando o get_dummies e colocando o prefixo = embarked_{nome || numero} e definindo o tipo de valor
- Adciono o dummies_embarked e dummies_pclass ao dados mas antes removo os `Embarked e Pclass`
- Removo valores `NaN` no df

In [ ]:
```
dummies_Embarked = pd.get_dummies(df['Embarked'],dtype=int,prefix='Embarked')
dummies_Pclass  =  pd.get_dummies(df['Pclass'],dtype=int,prefix='Pclass')
```

In [ ]:
```
df.drop(columns=['Embarked'],inplace=True)
df.drop(columns=['Pclass'],inplace=True)
df = pd.concat([df,dummies_Embarked],axis=1)
df = pd.concat([df,dummies_Pclass],axis=1)
```

In [ ]:
```
df.dropna(
    subset=['Sex','Age','SibSp','Parch','Fare',
            'Embarked_C','Embarked_Q','Embarked_S',
            'Pclass_1','Pclass_2','Pclass_3'],
    inplace=True
    )
df.shape
```

Out[ ]:
```
(714, 16)
```

Classificador

Escolhi alguns classficadores e comparei suas acuracias

- knn
- Regressao Logistica
- Decision Tree
- Naive bayes

In [ ]:
```
from sklearn.metrics         import confusion_matrix,ConfusionMatrixDisplay,accuracy_score,classification_report,roc_auc_score,
from sklearn.preprocessing   import LabelBinarizer,LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.linear_model    import LogisticRegression
from sklearn.naive_bayes     import GaussianNB
from sklearn.tree            import DecisionTreeClassifier
from sklearn.neighbors       import KNeighborsClassifier
```

In [ ]:
```
df['Sex'] = LabelBinarizer().fit_transform(df['Sex'])
```

In [ ]:
```
Y = df['Survived'].values
X = df[['Sex','Age','SibSp','Parch','Fare','Embarked_C','Embarked_Q','Embarked_S','Pclass_1','Pclass_2','Pclass_3']].values
```

In [ ]:
```
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.3,random_state=0)
```

Regressão Logistica

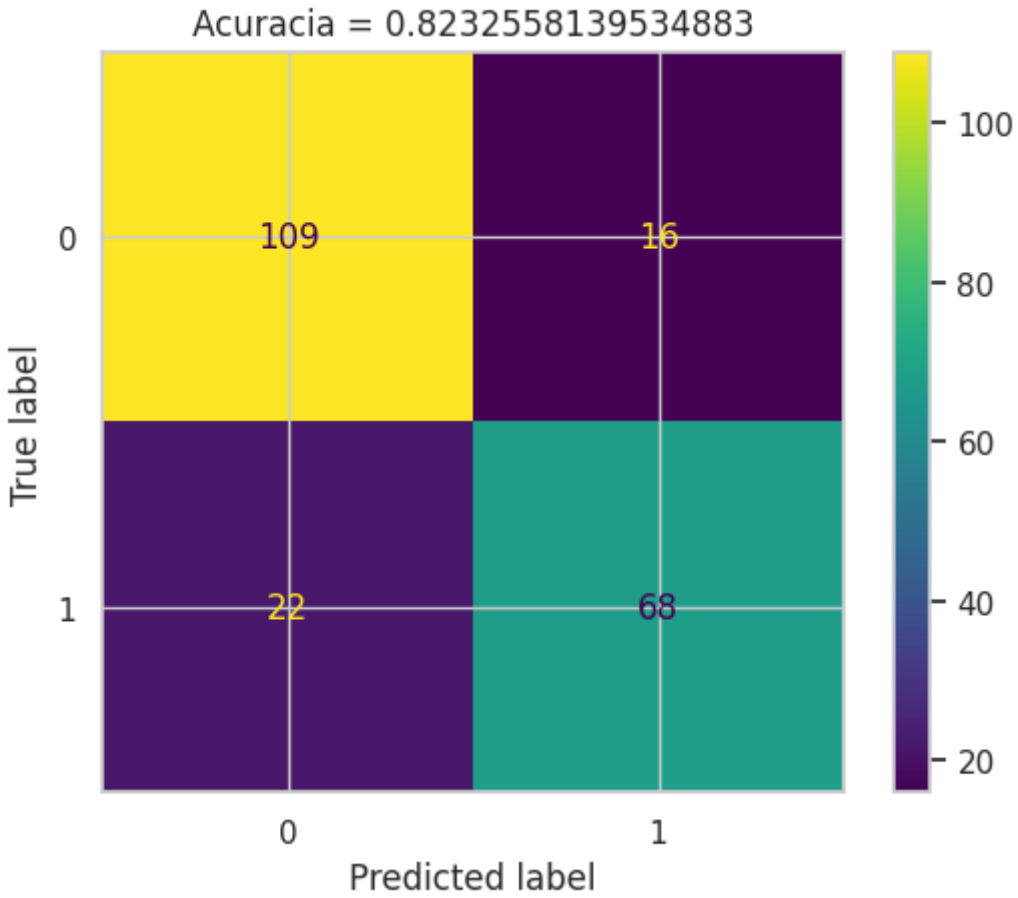
In [ ]:
```
regressaoLo = LogisticRegression(random_state=0,max_iter=100)
y_pred = regressaoLo.fit(x_train,y_train).predict(x_test)
```

```
/home/mateus/MEGA/Matérias UFC CD/Introdução a Mineração de Dados/venv/lib/python3.11/site-packages/sklearn/linear_model/_logist
ic.py:458: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

In [ ]:
```
regressaoLoAcuracia = accuracy_score(y_test,y_pred)
```

In [ ]:
```
cm = confusion_matrix(y_test,y_pred)

ConfusionMatrixDisplay(cm,display_labels=regressaoLo.classes_).plot()
plt.title(f"Acuracia = {regressaoLoAcuracia}")
```

Out[ ]:
```
Text(0.5, 1.0, 'Acuracia = 0.8232558139534883')
```



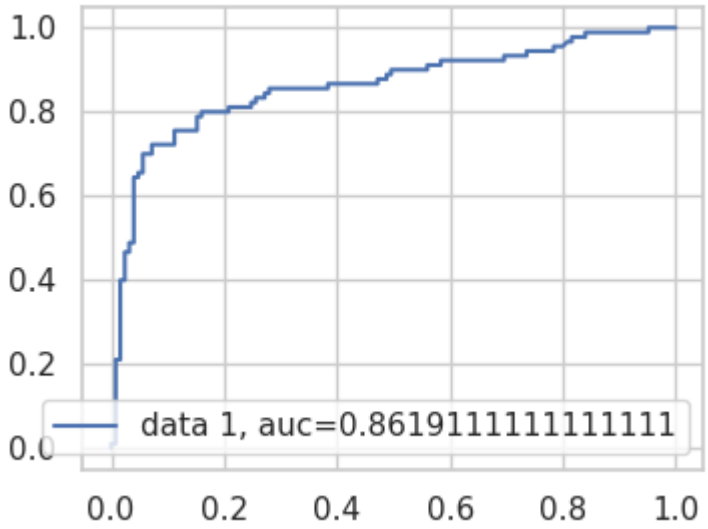
In [ ]:
```
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.83      0.87      0.85       125
           1       0.81      0.76      0.78        90

    accuracy                           0.82       215
   macro avg       0.82      0.81      0.82       215
weighted avg       0.82      0.82      0.82       215
```

In [ ]:
```
plt.figure(figsize=(4,3))
y_pred_prob    = regressaoLo.predict_proba(x_test)[::,1]
fpr, tpr, null = roc_curve(y_test,   y_pred_prob)
auc            = roc_auc_score(y_test,y_pred_prob)
```

```
plt.plot(fpr,tpr,label="data 1, auc="+str(auc))
plt.legend(loc=4)
```

Out[ ]: <matplotlib.legend.Legend at 0x7f44528b7c10>



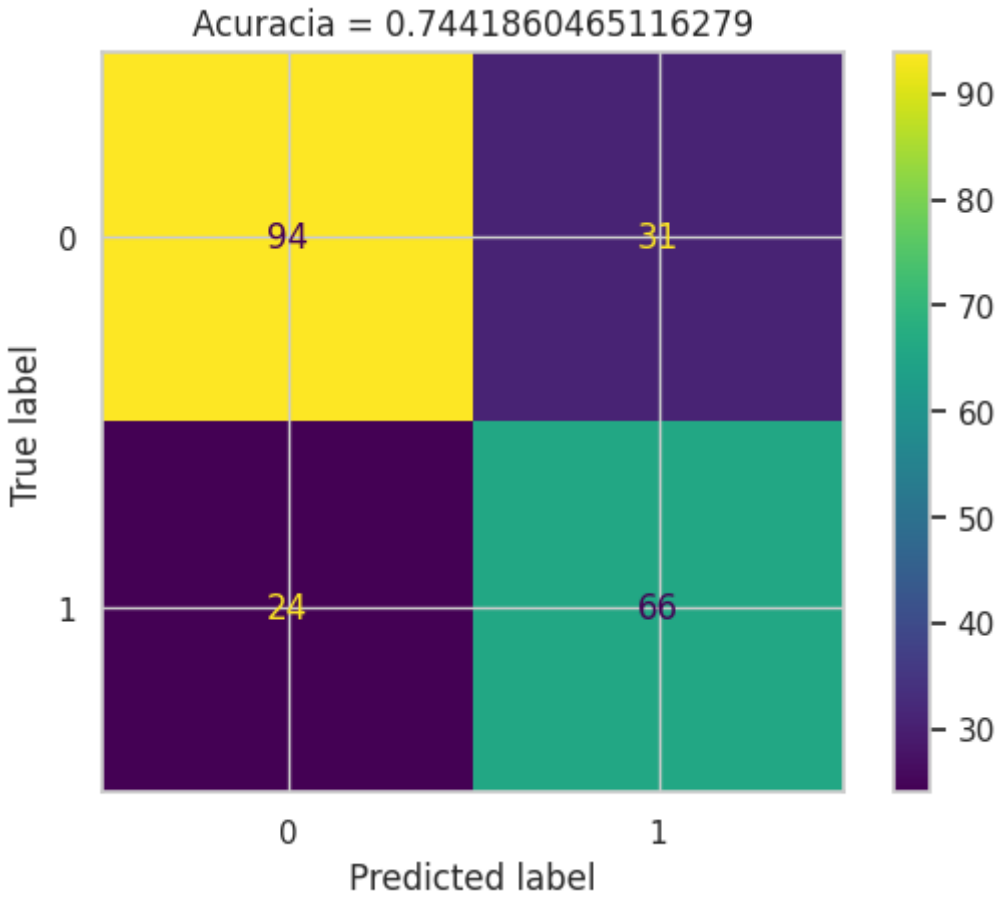
Arvore de decisão

In [ ]:
```
y_predTree = DecisionTreeClassifier(criterion='entropy').fit(x_train,y_train).predict(x_test)
```

In [ ]:
```
treeAcuracia = accuracy_score(y_test,y_predTree)
```

In [ ]:
```
cm = confusion_matrix(y_test,y_predTree)
ConfusionMatrixDisplay(cm,display_labels=regressaoLo.classes_).plot()
plt.title(f"Acuracia = {treeAcuracia}")
```

Out[ ]: Text(0.5, 1.0, 'Acuracia = 0.7441860465116279')



In [ ]:
```
print(classification_report(y_test,y_predTree))
```

```
              precision    recall  f1-score   support

           0       0.80      0.75      0.77       125
           1       0.68      0.73      0.71        90

    accuracy                           0.74       215
   macro avg       0.74      0.74      0.74       215
weighted avg       0.75      0.74      0.75       215
```

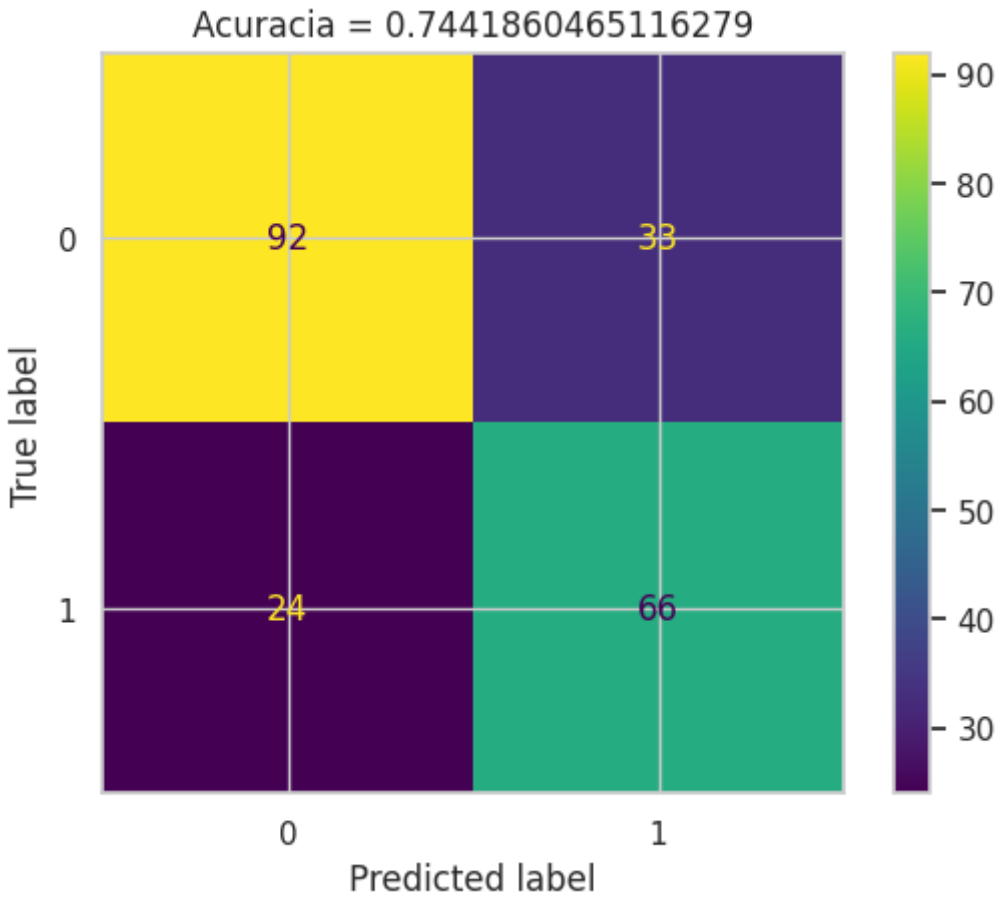
Naive bayes

In [ ]:
```
y_predBayes = GaussianNB().fit(x_train,y_train).predict(x_test)
```

In [ ]:
```
bayesAcuracia = accuracy_score(y_test,y_predBayes)
```

In [ ]:
```
cm = confusion_matrix(y_test,y_predBayes)

ConfusionMatrixDisplay(cm,display_labels=regressaoLo.classes_).plot()
plt.title(f"Acuracia = {treeAcuracia}")
```

Out[ ]:
```
Text(0.5, 1.0, 'Acuracia = 0.7441860465116279')
```



In [ ]:
```
print(classification_report(y_test,y_predBayes))
```

```
              precision    recall  f1-score   support

           0       0.79      0.74      0.76       125
           1       0.67      0.73      0.70        90

    accuracy                           0.73       215
   macro avg       0.73      0.73      0.73       215
weighted avg       0.74      0.73      0.74       215
```

KNN

In [ ]:
```
y_predKnn = KNeighborsClassifier().fit(x_train,y_train).predict(x_test)
```

In [ ]:
```
knnAcuracia = accuracy_score(y_test,y_predKnn)
```

In [ ]:
```
print(classification_report(y_test,y_predKnn))
```

```
              precision    recall  f1-score   support

           0       0.68      0.74      0.71       125
           1       0.59      0.52      0.56        90

    accuracy                           0.65       215
   macro avg       0.64      0.63      0.63       215
weighted avg       0.65      0.65      0.65       215
```

Comparando classificadores

In [ ]:
```
pd.DataFrame({
    'regressao Logistica':[regressaoLoAcuracia],
    'Arvores de Decisão ':[treeAcuracia],
    'Naives bayes'       :[bayesAcuracia],
    'KNN'                :[knnAcuracia]
},index=['accuracy_score'])
```

Out[ ]:

	regressao Logistica	Arvores de Decisão	Naives bayes	KNN
accuracy_score	0.823256	0.744186	0.734884	0.651163

> Regressão logistica obteve melhor resultado

Redes neurais

Testei varias redes. O que mais obtive resultado bom foi usando essa rede

In [ ]:
```
ann = ts.keras.models.Sequential()
```

In [ ]:
```
ann.add(ts.keras.layers.Flatten())
ann.add(ts.keras.layers.Dense(units= 12,activation='relu'))
ann.add(ts.keras.layers.Dense(units= 12,activation='relu'))
ann.add(ts.keras.layers.Dense(units= 8,activation='relu'))
ann.add(ts.keras.layers.Dense(units= 8,activation='relu'))
ann.add(ts.keras.layers.Dense(units= 1,activation='sigmoid'))
```

In [ ]:
```
ann.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
```

In [ ]:
```
treino = ann.fit(x_train, y_train, batch_size = 64 ,epochs = 100)
```

```
Epoch 1/100
8/8 [==============================] - 2s 4ms/step - loss: 0.9327 - accuracy: 0.5992
Epoch 2/100
8/8 [==============================] - 0s 3ms/step - loss: 0.7980 - accuracy: 0.5992
Epoch 3/100
8/8 [==============================] - 0s 3ms/step - loss: 0.7256 - accuracy: 0.6072
Epoch 4/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6680 - accuracy: 0.6313
Epoch 5/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6431 - accuracy: 0.6533
Epoch 6/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6330 - accuracy: 0.6653
Epoch 7/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6282 - accuracy: 0.6653
Epoch 8/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6253 - accuracy: 0.6713
Epoch 9/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6201 - accuracy: 0.6814
Epoch 10/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6140 - accuracy: 0.6954
Epoch 11/100
8/8 [==============================] - 0s 3ms/step - loss: 0.6079 - accuracy: 0.6934
Epoch 12/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5996 - accuracy: 0.6914
Epoch 13/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5955 - accuracy: 0.6954
Epoch 14/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5920 - accuracy: 0.6954
Epoch 15/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5899 - accuracy: 0.6934
Epoch 16/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5889 - accuracy: 0.6954
Epoch 17/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5838 - accuracy: 0.6994
Epoch 18/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5846 - accuracy: 0.6894
Epoch 19/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5802 - accuracy: 0.7014
Epoch 20/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5784 - accuracy: 0.6994
Epoch 21/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5756 - accuracy: 0.6954
Epoch 22/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5742 - accuracy: 0.7014
```

```
Epoch 23/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5706 - accuracy: 0.6894
Epoch 24/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5684 - accuracy: 0.6874
Epoch 25/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5652 - accuracy: 0.6974
Epoch 26/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5628 - accuracy: 0.6934
Epoch 27/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5571 - accuracy: 0.7034
Epoch 28/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5546 - accuracy: 0.7134
Epoch 29/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5521 - accuracy: 0.7275
Epoch 30/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5455 - accuracy: 0.7295
Epoch 31/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5417 - accuracy: 0.7214
Epoch 32/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5385 - accuracy: 0.7315
Epoch 33/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5316 - accuracy: 0.7335
Epoch 34/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5251 - accuracy: 0.7415
Epoch 35/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5210 - accuracy: 0.7415
Epoch 36/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5203 - accuracy: 0.7475
Epoch 37/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5126 - accuracy: 0.7435
Epoch 38/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5103 - accuracy: 0.7495
Epoch 39/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5075 - accuracy: 0.7535
Epoch 40/100
8/8 [==============================] - 0s 4ms/step - loss: 0.5054 - accuracy: 0.7515
Epoch 41/100
8/8 [==============================] - 0s 4ms/step - loss: 0.5059 - accuracy: 0.7555
Epoch 42/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5001 - accuracy: 0.7535
Epoch 43/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4967 - accuracy: 0.7675
Epoch 44/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4994 - accuracy: 0.7595
```

```
Epoch 45/100
8/8 [==============================] - 0s 3ms/step - loss: 0.5001 - accuracy: 0.7595
Epoch 46/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4915 - accuracy: 0.7595
Epoch 47/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4855 - accuracy: 0.7655
Epoch 48/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4842 - accuracy: 0.7675
Epoch 49/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4830 - accuracy: 0.7675
Epoch 50/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4813 - accuracy: 0.7675
Epoch 51/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4785 - accuracy: 0.7695
Epoch 52/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4757 - accuracy: 0.7756
Epoch 53/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4735 - accuracy: 0.7856
Epoch 54/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4710 - accuracy: 0.7936
Epoch 55/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4711 - accuracy: 0.7836
Epoch 56/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4683 - accuracy: 0.7816
Epoch 57/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4637 - accuracy: 0.7856
Epoch 58/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4702 - accuracy: 0.7856
Epoch 59/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4674 - accuracy: 0.7956
Epoch 60/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4673 - accuracy: 0.7916
Epoch 61/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4705 - accuracy: 0.7816
Epoch 62/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4714 - accuracy: 0.7856
Epoch 63/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4603 - accuracy: 0.7936
Epoch 64/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4597 - accuracy: 0.7916
Epoch 65/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4560 - accuracy: 0.7956
Epoch 66/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4506 - accuracy: 0.7876
```

```
Epoch 67/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4490 - accuracy: 0.7976
Epoch 68/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4482 - accuracy: 0.7956
Epoch 69/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4606 - accuracy: 0.7916
Epoch 70/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4697 - accuracy: 0.7715
Epoch 71/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4622 - accuracy: 0.7936
Epoch 72/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4501 - accuracy: 0.7856
Epoch 73/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4531 - accuracy: 0.7816
Epoch 74/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4427 - accuracy: 0.7856
Epoch 75/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4408 - accuracy: 0.7996
Epoch 76/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4408 - accuracy: 0.7976
Epoch 77/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4375 - accuracy: 0.7896
Epoch 78/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4369 - accuracy: 0.8016
Epoch 79/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4330 - accuracy: 0.8096
Epoch 80/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4302 - accuracy: 0.8076
Epoch 81/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4403 - accuracy: 0.8036
Epoch 82/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4383 - accuracy: 0.7996
Epoch 83/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4369 - accuracy: 0.7956
Epoch 84/100
8/8 [==============================] - 0s 5ms/step - loss: 0.4297 - accuracy: 0.7996
Epoch 85/100
8/8 [==============================] - 0s 6ms/step - loss: 0.4279 - accuracy: 0.8216
Epoch 86/100
8/8 [==============================] - 0s 5ms/step - loss: 0.4284 - accuracy: 0.7936
Epoch 87/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4242 - accuracy: 0.8216
Epoch 88/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4301 - accuracy: 0.7876
```

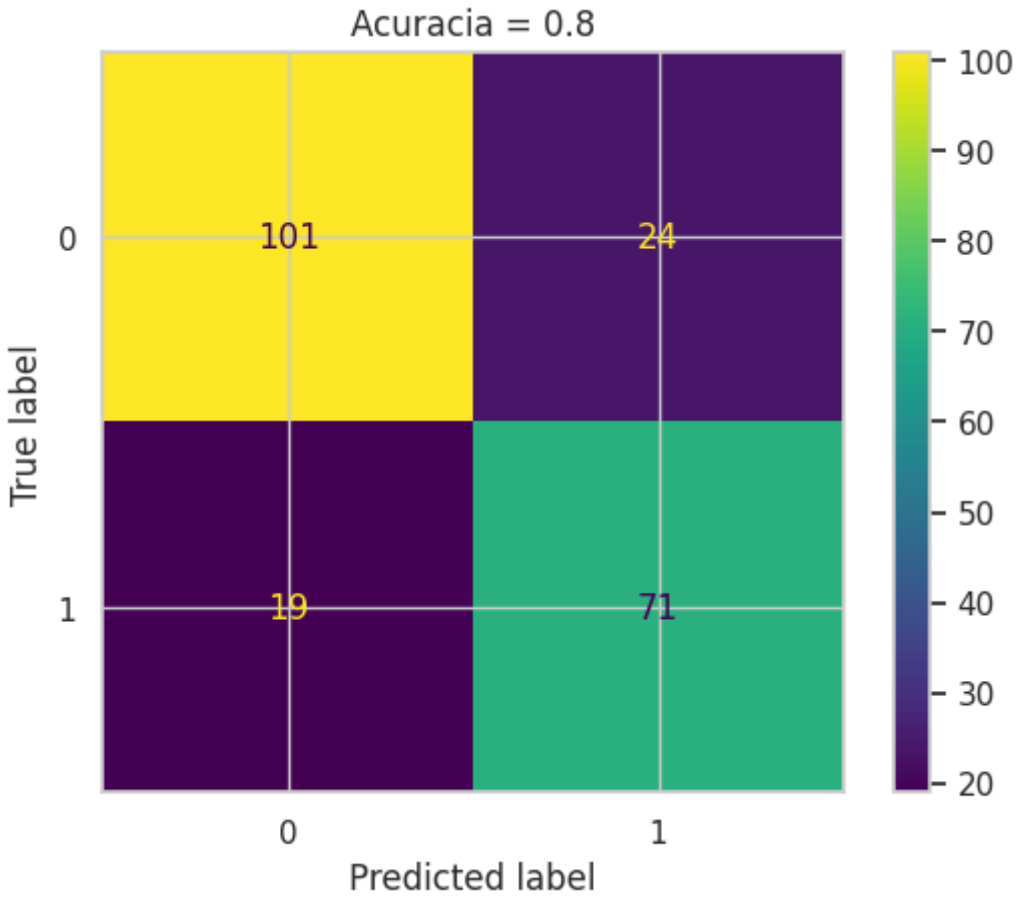
```
Epoch 89/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4231 - accuracy: 0.8096
Epoch 90/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4281 - accuracy: 0.7976
Epoch 91/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4317 - accuracy: 0.7996
Epoch 92/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4193 - accuracy: 0.8216
Epoch 93/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4187 - accuracy: 0.8156
Epoch 94/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4220 - accuracy: 0.8156
Epoch 95/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4233 - accuracy: 0.7996
Epoch 96/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4207 - accuracy: 0.8216
Epoch 97/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4174 - accuracy: 0.8036
Epoch 98/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4205 - accuracy: 0.8176
Epoch 99/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4161 - accuracy: 0.8076
Epoch 100/100
8/8 [==============================] - 0s 3ms/step - loss: 0.4171 - accuracy: 0.8297
```

In [ ]:
```
y_pred = ((ann.predict(x_test)>0.5)*1).T[0]
```

```
7/7 [==============================] - 0s 2ms/step
```

In [ ]:
```
cm = confusion_matrix(y_test,y_pred)

ConfusionMatrixDisplay(cm,display_labels=regressaoLo.classes_).plot()
plt.title(f"Acuracia = {accuracy_score(y_test,y_pred)}")
```

Out[ ]:
```
Text(0.5, 1.0, 'Acuracia = 0.8')
```


Acuracia = 0.8
True label
Predicted label
0
1
0
1
101
24
19
71
100
90
80
70
60
50
40
30
20