Tarefa-03. MO432.

Patrick de Carvalho Tavares Rezende Ferreira - 175480

In [1]:

```
import datetime

import numpy as np
from pandas import read_csv, get_dummies
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis, QuadraticD
iscriminantAnalysis
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import make_scorer, roc_auc_score
from sklearn.model_selection import ShuffleSplit, cross_validate, RandomizedSear
chCV, GridSearchCV
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier

%matplotlib inline
import warnings; warnings.simplefilter('ignore')
```

Leitura dos dados

Abaixo realizamos a leitura dos dados de entrada a partir do arquivo CSV, utilizando a API do "pandas". São removidas as colunas "Next_Tmin" e "Date", conforme solicitado no roteiro, além de todas as linhas que contenham valores faltantes ("nan").

Em seguida, separamos os dados de entrada da regressão ("X_data") e os dados alvo ("y_data"), fazendo *centering* e *scaling* na entrada em seguida.

In [2]:

```
# Obtem os dados do arquivo CSV.
df = read_csv("dados3.csv")
# # Elimina a coluna Next_Tmin.
# df = df.drop(columns=["Next_Tmin"])
# # Elimina a coluna Date
# df = df.drop(columns=["Date"])
# Elimina todas as linhas que contenham NAN (valor faltante).
df = df.dropna(axis=0, how='any')

get_dummies(df).to_csv("dados3-dummies.csv")

# OneHot encoding para converter vriaveis ctegoricas em dummy variables.
df = get_dummies(df)

# Passando os dados para um dataset numpy
y_data = df["V15"].to_numpy()
X_data = df.drop(columns="V15").to_numpy()

# Scaling dos dados em X.
scaler = StandardScaler()
scaler.fit(X_data)
X_data_scaled = scaler.transform(X_data)
```

Cross validation, medida de erro e busca de hiperparâmetros.

Usamos AUC como medida de score dos algoritmos de regressão, utilizando a repetição em 5-fold e buscando os hiperparâmetros utilizando o *random search* ou o *grid search* do pacote sklearn, a depender do exercício. Para comparar com os valores obtidos com o algoritmo padrão do sklearn, utilizamos o método *cross-validation*, que utiliza a validação cruzada sem desempenhar busca por qualquer parâmetro.

AUC da curva ROC

A curva ROC nos dá a relação entre a quantidade de Verdadeiros Positivos e Falsos positivos em função do limiar de decisão esccolhido. A AUC é a área debaixo desta curve e, logicamente, quanto maior, melhor será a distinção que o modelo proporciona na hora de classificar os dados em diferentes conjuntos.

Regressão Logística

Abaixo realizamos a regressão logística sem regulariazção, que não aplica hiperparâmetros e é o método de regressão mais rápido deste roteiro. O AUC médio das 5 repetições (folds) é de 0.9197457062275621.

In [3]:

```
# ============Logistic-Regression=========================================
np.random.seed(1234)

print("\n----------------Logistic-Regression----------------")

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = LogisticRegression(penalty="none", solver="lbfgs")
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"AUC": make_scorer(roc_auc_score, greater_is_better=
True, needs_proba=True)})

print("\nScore AUC parâmetros default: ", (cv_results["test_AUC"]).mean())
```

```
----------------Logistic-Regression----------------

Score AUC parâmetros default:  0.9197457062275621
```

Regressão Logística com regularização L2

Realizamos a regressão logística com regularização por norma L2 utilizando a API de classificadores do sklearn buscando o hiperparâmetro C de 10^{-3} a 10^{3}, uniforme no expoente. O melhor AUC obtido na média da validação cruzada é de 0.9407038551439606, para $C = 0.026020058428635535$, contra AUC de 0.9283532479273265 utilizando o C unitário default do sklearn. A diferença é pequena, mas o melhor resultado foi obtido com um pequeno valor de C possível na distribuição gerada, o que indica que este modelo não sofre de significativo overfitting, o que já se espera pelo fato de não utilizar funções não lineares.

In [4]:

```
# ============Logistic-Regression-L2=========================================
np.random.seed(3333)

# Gera os parametros de entrada aleatoriamente. Alguns sao uniformes nos
# EXPOENTES.
c = 10 ** np.random.uniform(-3, 3, 10)

# Une os parametros de entrada em um unico dicionario a ser passado para a
# funcao.
parametros = {'C': c}

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=3333)
regressor = LogisticRegression()
cv_results = \
    RandomizedSearchCV(estimator=regressor, cv=shuffle_splitter,
                       param_distributions=parametros,
                       refit="AUC",
                       verbose=1,
                       n_jobs=4,
                       scoring={"AUC": make_scorer(roc_auc_score, greater_is_bet
ter=True, needs_proba=True)})

# Realizamos a busca atraves do treinamento
cv_results.fit(X_data_scaled, y_data)

print("\n----------------Logistic-Regression-L2----------------")

print("\nMelhor conjunto de parâmetros: \n", cv_results.best_estimator_)

print("\nMelhor AUC score: \n", cv_results.best_score_, "\n")

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = LogisticRegression()
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"AUC": make_scorer(roc_auc_score, greater_is_better=
True, needs_proba=True)})

print("\nScore AUC parâmetros default: ", (cv_results["test_AUC"]).mean())
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits

[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent wo
rkers.

----------------Logistic-Regression-L2----------------

Melhor conjunto de parâmetros:
 LogisticRegression(C=0.026020058428635535)

Melhor AUC score:
 0.9405899735641581


Score AUC parâmetros default:  0.9282021419350347

[Parallel(n_jobs=4)]: Done  50 out of  50 | elapsed:    1.1s finishe
d
```

LDA

Abaixo realizamos a classificação com Linear Discriminant Analysis, que não aplica hiperparâmetros. O AUC médio das 5 repetições (folds) é de 0.9318615506427577. Como ele possui menos parâmetros a se ajustar que o QDA, seu resultado fica dentre os melhores do roteiro.

In [5]:

```
# ============LDA=================================================================
np.random.seed(1234)

print("\n----------------LDA----------------")

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = LinearDiscriminantAnalysis()
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"AUC": make_scorer(roc_auc_score, greater_is_better=
True, needs_proba=True)})

print("\nScore AUC parâmetros default: ", (cv_results["test_AUC"]).mean())
```

```
----------------LDA----------------

Score AUC parâmetros default:  0.9318615506427577
```

QDA

Abaixo realizamos a classificação com Quadratic Discriminant Analysis, que não aplica hiperparâmetros. O AUC médio das 5 repetições (folds) é de 0.8205157842023286. Mesmo sendo um método ainda mais flexível que o LDA, seus resultados são piores, e isto se deve ao fato que este modelo quadrático requer mais parâmetros a se ajustar. Em um dataset com menos de 1000 observações como este, o modelo fica subtreinado, obtendo um resultado de validação ruim.

In [6]:

```
# ============QDA==================================================================
np.random.seed(1234)

print("\n----------------QDA----------------")

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = QuadraticDiscriminantAnalysis()
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"AUC": make_scorer(roc_auc_score, greater_is_better=
True, needs_proba=True)})

print("\nScore AUC parâmetros default: ", (cv_results["test_AUC"]).mean())
```

```
----------------QDA----------------

Score AUC parâmetros default:  0.8205157842023286
```

SVC Linear

A busca por hiperparâmetros utilizando SVC com ativação linear retornou um AUC de 0.92809192351247, para $C = 31.925195621733018$, contra AUC de 0.9207682232859031 utilizando os parâmetros default. Estes são valores inferiores aos da regressão logística com L2 e descartam a utilização do SVR com ativação Linear para este tipo de problema, já que sua execução levou cerca de 10min, contra um resultado quase instantâneo da regressão.

In [7]:

```
# ============SVC-SVM-LINEAR=================================================
np.random.seed(3333)

# Gera os parametros de entrada aleatoriamente. Alguns sao uniformes nos
# EXPOENTES.
c = 2 ** np.random.uniform(-5, 15, 10)

# Une os parametros de entrada em um unico dicionario a ser passado para a
# funcao.
parametros = {'C': c}

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=3333)
regressor = SVC(max_iter=-1, cache_size=7000, kernel="linear", probability=True)
cv_results = \
    RandomizedSearchCV(estimator=regressor, cv=shuffle_splitter,
                       param_distributions=parametros,
                       refit="AUC",
                       verbose=1,
                       n_jobs=4,
                       scoring={"AUC": make_scorer(roc_auc_score, greater_is_bet
ter=True, needs_proba=True)})

# Realizamos a busca atraves do treinamento
cv_results.fit(X_data_scaled, y_data)

print("\n----------------SVC-SVM-LINEAR----------------")

print("\nMelhor conjunto de parâmetros: \n", cv_results.best_estimator_)

print("\nMelhor AUC score: \n", cv_results.best_score_, "\n")

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = SVC(max_iter=-1, cache_size=7000, kernel="linear", probability=True)
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"AUC": make_scorer(roc_auc_score, greater_is_better=
True, needs_proba=True)})

print("\nScore AUC parâmetros default: ", (cv_results["test_AUC"]).mean())
```

```
[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent wo
rkers.

Fitting 5 folds for each of 10 candidates, totalling 50 fits

[Parallel(n_jobs=4)]: Done  42 tasks      | elapsed:  5.1min
[Parallel(n_jobs=4)]: Done  50 out of  50 | elapsed:  6.9min finishe
d

----------------SVC-SVM-LINEAR---------------

Melhor conjunto de parâmetros:
 SVC(C=31.925195621733018, cache_size=7000, kernel='linear', probabi
lity=True)

Melhor AUC score:
 0.9280919235124699


Score AUC parâmetros default:  0.9207496953677872
```

SVC RBF

A busca por hiperparâmetros utilizando SVC com ativação RBF (Radial basis function) retornou um AUC de 0.9336317703698913, para $C = 0.16414560961711494$ e $Gamma = 0.00699943241971803$, contra AUC de 0.9342224478698687 utilizando os parâmetros default. Estes são valores inferiores aos da regressão logística com L2 e descartam a utilização do SVR com RBF para este tipo de problema, já que sua execução levou cerca de 10min, contra um resultado quase instantâneo da regressão.

Este resultado também demonstra que o default do sklearn é bem ajustado o suficiente, já que produziu resultado semelhante ao da busca.

In [8]:

```
# ============SVC-SVM-RBF=========================================================
np.random.seed(3333)

# Gera os parametros de entrada aleatoriamente. Alguns sao uniformes nos
# EXPOENTES.
c = 2 ** np.random.uniform(-5, 15, 10)
gamma = 2 ** np.random.uniform(-9, 3, 10)

# Une os parametros de entrada em um unico dicionario a ser passado para a
# funcao.
parametros = {'C': c, 'gamma': gamma}

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=3333)
regressor = SVC(max_iter=-1, cache_size=7000, kernel="rbf", probability=True)
cv_results = \
    RandomizedSearchCV(estimator=regressor, cv=shuffle_splitter,
                       param_distributions=parametros,
                       refit="AUC",
                       verbose=1,
                       n_jobs=4,
                       scoring={"AUC": make_scorer(roc_auc_score, greater_is_bet
ter=True, needs_proba=True)})

# Realizamos a busca atraves do treinamento
cv_results.fit(X_data_scaled, y_data)

print("\n----------------SVC-SVM-RBF----------------")

print("\nMelhor conjunto de parâmetros: \n", cv_results.best_estimator_)

print("\nMelhor AUC score: \n", cv_results.best_score_, "\n")

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = SVC(max_iter=-1, cache_size=7000, kernel="rbf", probability=True)
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"AUC": make_scorer(roc_auc_score, greater_is_better=
True, needs_proba=True)})

print("\nScore AUC parâmetros default: ", (cv_results["test_AUC"]).mean())
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits

[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent wo
rkers.
[Parallel(n_jobs=4)]: Done  50 out of  50 | elapsed:    1.6s finishe
d

----------------SVC-SVM-RBF---------------

Melhor conjunto de parâmetros:
 SVC(C=0.16414560961711494, cache_size=7000, gamma=0.006999432419718
03,
    probability=True)

Melhor AUC score:
 0.9336222210956361


Score AUC parâmetros default:  0.9341840620406815
```

Naive Bayes

Abaixo utilizamos o método Naives Bayes com GaussianNB, que computa as ocorrências de cada combinação para estimar a probabiidade e a condicional de cada evento, além de fazer a predição com base no teorema de Bayes. O AUC obtido foi de 0.8632802184759667, um dos piores do roteiro, provavelmente pela pequena quantidade de dados do dataset.

In [9]:

```
# ============GaussianNB-Naive-Bayes=======================================

print("\n----------------GaussianNB-Naive-Bayes----------------")

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = GaussianNB()
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"AUC": make_scorer(roc_auc_score, greater_is_better=
True, needs_proba=True)})

print("\nScore AUC parâmetros default: ", (cv_results["test_AUC"]).mean())
```

```
----------------GaussianNB-Naive-Bayes---------------

Score AUC parâmetros default:  0.8632802184759667
```

KNN

Na célula abaixo, realizamos a classificação por meio do "*K-nearest neighbors*" classificador, que seleciona os "k" valores mais próximos do dado a ser amostrado dentre os dados passados para aprendizado e retorna uma classe que pode ser ponderada em seus votos em função da distância de cada um. Nota-se que o AUC obtido pelo melhor parâmetro encontrado (k=187 vizinhos) é de 0.9273362726256549, enquanto que o AUC dos parâmetros default do sklearn foi de 0.8759436830922536. É um dos melhores métodos até agora, mas não superou a regressão logística.

In [10]:

```
# ============KNeighborsRegressor=============================================
np.random.seed(1234)

# Gera os parametros de entrada aleatoriamente.
n_neighbors = np.random.uniform(0, 150, 10).astype("int32") * 2 + 1

# Une os parametros de entrada em um unico dicionario a ser passado para a
# funcao.
parametros = {'n_neighbors': n_neighbors}

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = KNeighborsClassifier()
cv_results = \
    RandomizedSearchCV(estimator=regressor, cv=shuffle_splitter,
                       param_distributions=parametros,
                       refit="AUC",
                       verbose=1,
                       n_jobs=4,
                       scoring={"AUC": make_scorer(roc_auc_score, greater_is_bet
ter=True, needs_proba=True)})

# Realizamos a busca atraves do treinamento
cv_results.fit(X_data_scaled, y_data)

print("\n----------------KNeighborsClassifier----------------")

print("\nMelhor conjunto de parâmetros: \n", cv_results.best_estimator_)

print("\nMelhor AUC score: \n", cv_results.best_score_, "\n")

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = KNeighborsClassifier()
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"AUC": make_scorer(roc_auc_score, greater_is_better=
True, needs_proba=True)})

print("\nScore AUC parâmetros default: ", (cv_results["test_AUC"]).mean())
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits

[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent wo
rkers.

----------------KNeighborsClassifier----------------

Melhor conjunto de parâmetros:
 KNeighborsClassifier(n_neighbors=187)

Melhor AUC score:
 0.9273362726256549


Score AUC parâmetros default:  0.8759436830922536

[Parallel(n_jobs=4)]: Done  50 out of  50 | elapsed:    0.3s finishe
d
```

MLP

Utilizando a MLP para classificação, fica clara a atuação do teorema da aproximação universal, que prevê que uma MLP com uma única camada oculta é capaz de aproximar qualquer função contínua se forem fornecidos suficientes neurônios para a camada oculta, bem como épocas ou iterações do treino. Isto se mostra no fato que o classificador encontrou que 17 neurônios oferecidos para a camada oculta durante o treinamento produzia a melhor AUC, de 0.9273499770680775, enquanto que o default do sklearn produziu um AUC inferior, de 0.9142395181818566, por utilizar 100 neurônios na camada oculta. Lembrando que adicionar neurônios demais para poucos dados implica em ter parâmetros subajustados, o que explica os resultaos inferiores no default com mais neurônios.

In [11]:

```
# ============MLPClassifier=====================================================
np.random.seed(1234)

# Gera os parametros de entrada aleatoriamente.
hidden_layer_sizes = np.array(range(5, 21, 3))

# Une os parametros de entrada em um unico dicionario a ser passado para a
# funcao.
parametros = {'hidden_layer_sizes': hidden_layer_sizes}

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = MLPClassifier()
cv_results = \
    RandomizedSearchCV(estimator=regressor, cv=shuffle_splitter,
                       param_distributions=parametros,
                       refit="AUC",
                       verbose=1,
                       n_jobs=4,
                       scoring={"AUC": make_scorer(roc_auc_score, greater_is_bet
ter=True, needs_proba=True)})

# Realizamos a busca atraves do treinamento
cv_results.fit(X_data_scaled, y_data)

print("\n---------------MLPClassifier-------------------")

print("\nMelhor conjunto de parâmetros: \n", cv_results.best_estimator_)

print("\nMelhor AUC score: \n", cv_results.best_score_, "\n")

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = MLPClassifier()
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"AUC": make_scorer(roc_auc_score, greater_is_better=
True, needs_proba=True)})

print("\nScore AUC parâmetros default: ", (cv_results["test_AUC"]).mean())
```

```
[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent wo
rkers.

Fitting 5 folds for each of 6 candidates, totalling 30 fits

[Parallel(n_jobs=4)]: Done  30 out of  30 | elapsed:    3.8s finishe
d

---------------MLPClassifier-------------------

Melhor conjunto de parâmetros:
 MLPClassifier(hidden_layer_sizes=17)

Melhor AUC score:
 0.9289030823173332


Score AUC parâmetros default:  0.9142395181818566
```

Árvore de decisão

Ao se utilizar uma única árvore de decisão com prunning variável (ccp_alpha é o hiperparâmetro sendo buscado), obtivemos 0.9091774473094114 como AUC do melhor ccp_alpha, sendo este de 0.01750910956028458. O resultado foi melhor do que o default do sklearn, que obteve AUC de 0.8272019408426212 com ccp_alpha de zero, ou seja, sem prunning.

In [12]:

```
# ============DecisionTreeClassifier=========================================
np.random.seed(1234)

# Gera os parametros de entrada aleatoriamente.
ccp_alpha = np.random.uniform(0, 0.04, 10)

# Une os parametros de entrada em um unico dicionario a ser passado para a
# funcao.
parametros = {'ccp_alpha': ccp_alpha}

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = DecisionTreeClassifier()
cv_results = \
    RandomizedSearchCV(estimator=regressor, cv=shuffle_splitter,
                       param_distributions=parametros,
                       refit="AUC",
                       verbose=1,
                       n_jobs=4,
                       scoring={"AUC": make_scorer(roc_auc_score, greater_is_bet
ter=True, needs_proba=True)})

# Realizamos a busca atraves do treinamento
cv_results.fit(X_data_scaled, y_data)

print("\n--------------DecisionTreeClassifier------------------")

print("\nMelhor conjunto de parâmetros: \n", cv_results.best_estimator_)

print("\nMelhor AUC score: \n", cv_results.best_score_, "\n")

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = DecisionTreeClassifier()
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"AUC": make_scorer(roc_auc_score, greater_is_better=
True, needs_proba=True)})

print("\nScore AUC parâmetros default: ", (cv_results["test_AUC"]).mean())
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits

--------------DecisionTreeClassifier------------------

Melhor conjunto de parâmetros:
 DecisionTreeClassifier(ccp_alpha=0.01750910956028458)

Melhor AUC score:
 0.9091774473094114


Score AUC parâmetros default:  0.8272019408426212

[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent wo
rkers.
[Parallel(n_jobs=4)]: Done  50 out of  50 | elapsed:    0.1s finishe
d
```

Random Forest

Utilizando o método Random Forest, que une múltiplas árvores de decisão por meio do voto majoritário, obtivemos um ganho elevado no AUC, sendo de 0.943294639551229 com 1000 estimadores e no máximo 10 features, contra 0.9395490611060222 utilizando os parâmetros default do sklearn, sendo ambos consideravelmente melhores que os demais métodos. Esta busca por hiperparâmetros mostrou que aumentar o número de árvores foi algo bom, assim como aumentar a quantidade de features.

O único contra deste método em relação ao anterior é a perda da possível interpretabilidade do método, já que não é mais explicável o que está sendo feito quando se tem um número grande de árvores atuando em conjunto, como é o caso.

In [13]:

```
# ============RandomForestClassifier=========================================
np.random.seed(1234)

# Gera os parametros de entrada aleatoriamente.
n_estimators = [10, 100, 1000]
max_features = [5, 8, 10]

# Une os parametros de entrada em um unico dicionario a ser passado para a
# funcao.
parametros = {'n_estimators': n_estimators, 'max_features': max_features}

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = RandomForestClassifier()
cv_results = \
    GridSearchCV(estimator=regressor, cv=shuffle_splitter,
                 param_grid=parametros,
                 verbose=1,
                 refit="AUC",
                 n_jobs=1,
                 scoring={"AUC": make_scorer(roc_auc_score, greater_is_better=Tr
ue, needs_proba=True)})

# Realizamos a busca atraves do treinamento
cv_results.fit(X_data_scaled, y_data)

print("\n--------------RandomForestClassifier------------------")

print("\nMelhor conjunto de parâmetros: \n", cv_results.best_estimator_)

print("\nMelhor AUC score: \n", cv_results.best_score_, "\n")

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = RandomForestClassifier()
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"AUC": make_scorer(roc_auc_score, greater_is_better=
True, needs_proba=True)})

print("\nScore AUC parâmetros default: ", (cv_results["test_AUC"]).mean())
```

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurr
ent workers.
[Parallel(n_jobs=1)]: Done  45 out of  45 | elapsed:   27.8s finishe
d

--------------RandomForestClassifier------------------

Melhor conjunto de parâmetros:
 RandomForestClassifier(max_features=10, n_estimators=1000)

Melhor AUC score:
 0.943294639551229


Score AUC parâmetros default:  0.9395490611060222
```

GBM

Por último, trabalhamos com o GBM (Gradient Boosting Classifier), que adiciona modelos treinados com diferentes pesos nas regiões de maior erro do anterior, de forma que os sistemas sendo adicionados possam treinar sobre as regiões onde os modelos anteriores mais erraram.

O AUC obtido com os melhores parâmetos encontrados foi de 0.9412940669742994, sendo os parâmetros de: n_estimators = 96, learning_rate = 0.01 e max_depth = 2. Pode-se concluir que mais estimadores tornam o modelo melhor, o learning rate de 0.01 demonstrou uma precisão melhor no ajuste do modelo e uma profundidade menor na árvore foi suficiente a classficação.O AUC obtido com o default foi de 0.937917689530414 9

In [14]:

```
# ============GradientBoostingClassifier=====================================
np.random.seed(1234)

# Gera os parametros de entrada aleatoriamente.
n_estimators = np.random.uniform(5, 100, 10).astype("int32")
learning_rate = [0.01, 0.3]
max_depth = [2, 3]

# Une os parametros de entrada em um unico dicionario a ser passado para a
# funcao.
parametros = {'n_estimators': n_estimators, 'learning_rate': learning_rate, 'max
_depth': max_depth}

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = GradientBoostingClassifier()
cv_results = \
    RandomizedSearchCV(estimator=regressor, cv=shuffle_splitter,
                       param_distributions=parametros,
                       refit="AUC",
                       verbose=1,
                       n_jobs=4,
                       scoring={"AUC": make_scorer(roc_auc_score, greater_is_bet
ter=True, needs_proba=True)})

# Realizamos a busca atraves do treinamento
cv_results.fit(X_data_scaled, y_data)

print("\n--------------GradientBoostingClassifier------------------")

print("\nMelhor conjunto de parâmetros: \n", cv_results.best_estimator_)

print("\nMelhor AUC score: \n", cv_results.best_score_, "\n")

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = GradientBoostingClassifier()
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"AUC": make_scorer(roc_auc_score, greater_is_better=
True, needs_proba=True)})

print("\nScore AUC parâmetros default: ", (cv_results["test_AUC"]).mean())
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits

[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent wo
rkers.
[Parallel(n_jobs=4)]: Done  50 out of  50 | elapsed:    1.8s finishe
d


--------------GradientBoostingClassifier------------------

Melhor conjunto de parâmetros:
 GradientBoostingClassifier(learning_rate=0.01, max_depth=2, n_estim
ators=96)

Melhor AUC score:
 0.9412940669742994


Score AUC parâmetros default:  0.9379176895304149
```

Conclusão

Random Forest, GBM e Logistic Regression com regularização L2 foram os métodos com melhores resultados deste roteiro, respectivamente. O AUC obtido por cada um deles foi praticamente o mesmo, diferindo por milésimos.

Entretanto, a regressão logística foi consideravelmente mais rápida, podendo ser um método preferível na maioria dos casos.

Também ficou evidente o fato que não é conveniente o modelo ter mais flexibilidade se o tamanho do dataset para treiná-lo não é grande o suficiente, pois produz parâmetros subajustados que desempenham mal as tarefas de validação.