# Tarefa-02. MO432.

## Patrick de Carvalho Tavares Rezende Ferreira - 175480

In [1]:

```
import random

import numpy as np
from pandas import read_csv
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.metrics import make_scorer, mean_squared_error, mean_absolute_error
from sklearn.model_selection import ShuffleSplit, cross_validate, RandomizedSear
chCV, GridSearchCV
from sklearn.neighbors import KNeighborsRegressor
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVR
from sklearn.tree import DecisionTreeRegressor

%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("Bias_correction_ucl.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')

# Passando os dados para um dataset numpy
y_data = df["Next_Tmax"].to_numpy()
X_data = df.drop(columns="Next_Tmax").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 RMSE como medida de erro 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.

## Regressão Linear

Abaixo realizamos a regressão linear por método dos mínimos quadrados, que não aplica hiperparâmetros e é o método de regressão mais rápido deste roteiro. O erro RMSE médio das 5 repetições (folds) é de 1.4668565460537988°C.

In [3]:

```
# ============LINEAR-REGRESSION=================================================

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = LinearRegression()
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"MSE": make_scorer(mean_squared_error, greater_is_be
tter=False),
                            "MAE": make_scorer(mean_absolute_error, greater_is_b
etter=False)})

print("\n--------------------LINEAR_REGRESSION--------------------")

print("\nRMSE para cada repetição: \n", (-cv_results["test_MSE"]) ** (1 / 2))

print("\n\nRMSE médio: ", ((-cv_results["test_MSE"]) ** (1 / 2)).mean())
```

```
--------------------LINEAR_REGRESSION--------------------

RMSE para cada repetição:
 [1.49934913 1.47853876 1.45404056 1.4510827  1.45127158]


RMSE médio:  1.4668565460537988
```

## Regressão linear com regularização L2

Realizamos a regressão linear com regularização por norma L2 (Ridge regression) utilizando a API de regressores do sklearn buscando o hiperparâmetro alpha de $10^{-3}$ a $10^{3}$, uniforme no expoente. O melhor RMSE obtido na média da validação cruzada é de 1.4668565552684352°C, para $\alpha = 0.001$, contra RMSE de 1.4668660324711131°C utilizando o alpha unitário default do sklearn. A diferença é pequena, mas o melhor resultado foi obtido com o menor valor de alpha possível, 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]:

```
# ============L2-RIDGE-REGRESSION=============================================

np.random.seed(1234)

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

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

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = Ridge()
cv_results = \
    RandomizedSearchCV(estimator=regressor, cv=shuffle_splitter,
                       param_distributions=parametros,
                       verbose=1,
                       n_jobs=4,
                       scoring="neg_root_mean_squared_error")

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

print("\n--------------------LINEAR_REGRESSION_L2-------------------")

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

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

# Deafult do sklearn. Coloquei uma lista de 10 parametros iguais so pra nao dar
 warning, performance nao eh critico aqui
alpha = [1.0] * 10

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

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = Ridge()
cv_results = \
    RandomizedSearchCV(estimator=regressor, cv=shuffle_splitter,
                       param_distributions=parametros,
                       verbose=1,
                       n_jobs=4,
                       scoring="neg_root_mean_squared_error")

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

print("\nScore RMSE default do sklearn: \n", -cv_results.best_score_)
```

```
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:    0.9s finishe
d
[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent wo
rkers.

---------------------LINEAR_REGRESSION_L2-------------------

Melhor conjunto de parâmetros:
 Ridge(alpha=0.001, copy_X=True, fit_intercept=True, max_iter=None,
      normalize=False, random_state=None, solver='auto', tol=0.001)

Melhor error score:
 1.4668565552684352
Fitting 5 folds for each of 10 candidates, totalling 50 fits

Score RMSE default do sklearn:
 1.4668660324711131

[Parallel(n_jobs=4)]: Done  50 out of  50 | elapsed:    0.2s finishe
d
```

## Regressão linear com regularização L1

Realizamos a regressão linear com regularização por norma L1 (Lasso regression) utilizando a API de regressores do sklearn buscando o hiperparâmetro alpha de $10^{-3}$ a $10^{3}$, uniforme no expoente. O melhor RMSE obtido na média da validação cruzada é de 1.4668877424692912°C, para $\alpha = 0.001$, contra RMSE de 1.9815798242208555°C utilizando o alpha unitário default do sklearn. A diferença é de mais de 30%, e o melhor resultado foi obtido com o menor valor de alpha possível, o que indica que este modelo não sofre de significativo overfitting, o que já se espera pelo fato de não utilizar funções não lineares. Além disso, a diferença entre o modelo com melhor resultado (baixa regularização) contra o default (maior peso na regularização), indica que a norma L1 prejudica o aprendizado neste caso, provavelmente reduzindo drasticamente o peso sobre importantes dados de entrada.

In [5]:

```
# ============L1-LASSO-REGRESSION=============================================

np.random.seed(1234)

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

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

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = Lasso()
cv_results = \
    RandomizedSearchCV(estimator=regressor, cv=shuffle_splitter,
                       param_distributions=parametros,
                       verbose=1,
                       n_jobs=4,
                       scoring="neg_root_mean_squared_error")

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

print("\n--------------------LINEAR_REGRESSION_L1------------------")

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

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

# Deafult do sklearn. Coloquei uma lista de 10 parametros iguais so pra nao dar
 warning, performance nao eh critico aqui
alpha = [1.0] * 10

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

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = Lasso()
cv_results = \
    RandomizedSearchCV(estimator=regressor, cv=shuffle_splitter,
                       param_distributions=parametros,
                       verbose=1,
                       n_jobs=4,
                       scoring="neg_root_mean_squared_error")

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

print("\nScore RMSE default do sklearn: \n", -cv_results.best_score_)
```

```
[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  50 out of  50 | elapsed:    0.2s finishe
d
[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent wo
rkers.

---------------------LINEAR_REGRESSION_L1------------------

Melhor conjunto de parâmetros:
 Lasso(alpha=0.001, copy_X=True, fit_intercept=True, max_iter=1000,
      normalize=False, positive=False, precompute=False, random_stat
e=None,
      selection='cyclic', tol=0.0001, warm_start=False)

Melhor error score:
 1.4668877424692912
Fitting 5 folds for each of 10 candidates, totalling 50 fits

Score RMSE default do sklearn:
 1.9815798242208555

[Parallel(n_jobs=4)]: Done  50 out of  50 | elapsed:    0.2s finishe
d
```

## SVR Linear

A busca por hiperparâmetros utilizando SVR com ativação linear retornou um RMSE de 1.4556835168247064°C, para $\epsilon = 0.1$ e $c = 5.46874897339475$, contra RMSE de 1.4713019150729336 utilizando os parâmetros default. Estes são valores mutio próximos aos obtidos pela regressão linear comum e descartam a utilização do SVR com ativação Linear para este tipo de problema, já que sua execução levou cerca de 2h, contra um resultado quase instantâneo da regressão linear.

In [6]:

```
# ============SVR-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)
epsilon = np.array(random.choices([0.1, 0.3], k=10))

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

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=3333)
regressor = SVR(max_iter=-1, cache_size=7000, kernel="linear")
cv_results = \
    RandomizedSearchCV(estimator=regressor, cv=shuffle_splitter,
                       param_distributions=parametros,
                       verbose=1,
                       n_jobs=4,
                       scoring="neg_root_mean_squared_error")

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

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

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

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

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = SVR(max_iter=-1, cache_size=7000, kernel="linear")
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"MSE": make_scorer(mean_squared_error, greater_is_be
tter=False),
                            "MAE": make_scorer(mean_absolute_error, greater_is_b
etter=False)})

print("\nScore RMSE parâmetros default: ", ((-cv_results["test_MSE"]) ** (1 / 2
)).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  42 tasks      | elapsed: 71.0min
[Parallel(n_jobs=4)]: Done  50 out of  50 | elapsed: 80.9min finishe
d

----------------SVR-SVM-LINEAR---------------

Melhor conjunto de parâmetros:
 SVR(C=1046.612837141013, cache_size=7000, coef0=0.0, degree=3, epsi
lon=0.1,
    gamma='scale', kernel='linear', max_iter=-1, shrinking=True, tol
=0.001,
    verbose=False)

Melhor error score:
 1.4556151839072504

Score RMSE parâmetros default:  1.4713019150729336
```

## SVR com kernel RBF

Este foi o regressor que obteve o melhor RMSE do roteiro, sendo de 0.9407269250783268°C para $c = 5.46874897339475$, $\epsilon = 0.1$ e $\gamma = 0.08185402239753949$, contra RMSE de 1.193287479822758°C para os parâmetros default do sklearn. O tempo de treinamento para convergir é elevado, sendo que o processo de busca levou cerca de 2h, indicando que este método vale a pena se houver necessidade de um desempenho extremamente elevado e houver tempo disponível.

In [7]:

```
# ============SVR-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)
epsilon = np.array(random.choices([0.1, 0.3], k=10))

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

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=3333)
regressor = SVR(max_iter=-1, cache_size=7000, kernel="rbf")
cv_results = \
    RandomizedSearchCV(estimator=regressor, cv=shuffle_splitter,
                       param_distributions=parametros,
                       verbose=1,
                       n_jobs=4,
                       scoring="neg_root_mean_squared_error")

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

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

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

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

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = SVR(max_iter=-1, cache_size=7000, kernel="rbf")
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"MSE": make_scorer(mean_squared_error, greater_is_be
tter=False),
                            "MAE": make_scorer(mean_absolute_error, greater_is_b
etter=False)})

print("\nScore RMSE parâmetros default: ", ((-cv_results["test_MSE"]) ** (1 / 2
)).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  42 tasks      | elapsed:  1.7min
[Parallel(n_jobs=4)]: Done  50 out of  50 | elapsed:  1.8min finishe
d

----------------SVR-SVM-RBF----------------

Melhor conjunto de parâmetros:
 SVR(C=282.51819623608304, cache_size=7000, coef0=0.0, degree=3, eps
ilon=0.3,
    gamma=0.012079992323024893, kernel='rbf', max_iter=-1, shrinking
=True,
    tol=0.001, verbose=False)

Melhor error score:
 0.9384120100229417

Score RMSE parâmetros default:  1.193287479822758
```

## KNN

Na célula abaixo, realizamos a regressão por meio do "*K-nearest neighbors*" regressor, que seleciona os "k" valores mais próximos do dado a ser amostrado dentre os dados passados para aprendizado e retorna uma média que pode ser ponderada em função da distância de cada um. Nota-se que o erro obtido pelo melhor parâmetro encontrado (k=192 vizinhos) é de 1.740118737742985°C, enquanto que o erro RMSE dos parâmetros default do sklearn foi de 1.2702952951681985°C. Isto provavelmente se deve ao fato que o melhor valor para "k" seria um número pequeno, mas como amostramos apenas 10 números aleatórios entre 1 e 1000 para "k", provavelmente não obtivemos um valor de vizinhos que superasse o default do método. Uma sugestão é fazer uma busca refinada entre k=5 (default) e k=192 (melhor do random search).

In [8]:

```
# ============KNeighborsRegressor=============================================

np.random.seed(1234)

# Gera os parametros de entrada aleatoriamente.
n_neighbors = np.linspace(1, 1000, 10).astype("int32")

# 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 = KNeighborsRegressor()
cv_results = \
    RandomizedSearchCV(estimator=regressor, cv=shuffle_splitter,
                       param_distributions=parametros,
                       verbose=1,
                       n_jobs=4,
                       scoring="neg_root_mean_squared_error")

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

print("\n----------------KNeighborsRegressor----------------")

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

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

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = KNeighborsRegressor()
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"MSE": make_scorer(mean_squared_error, greater_is_be
tter=False),
                            "MAE": make_scorer(mean_absolute_error, greater_is_b
etter=False)})

print("\nScore RMSE parâmetros default: ", ((-cv_results["test_MSE"]) ** (1 / 2
)).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  42 tasks      | elapsed:   16.7s
[Parallel(n_jobs=4)]: Done  50 out of  50 | elapsed:   19.9s finishe
d

----------------KNeighborsRegressor---------------

Melhor conjunto de parâmetros:
 KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkows
ki',
                    metric_params=None, n_jobs=None, n_neighbors=1,
p=2,
                    weights='uniform')

Melhor error score:
 1.619543068372905

Score RMSE parâmetros default:  1.2702952951681985
```

## MLP

Utilizando a MLP para regressão, 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 regressor encontrou que 20 (máximo de) neurônios oferecidos para a camada oculta durante o treinamento produzia a melhor RMSE, de 1.9043734674080024°C, enquanto que o default do sklearn produziu uma RMSE ainda melhor, de 1.2823751962533572°C, por utilizar 100 neurônios na camada oculta.

In [9]:

```
# ============MLPRegressor=====================================================

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 = MLPRegressor()
cv_results = \
    RandomizedSearchCV(estimator=regressor, cv=shuffle_splitter,
                       param_distributions=parametros,
                       verbose=1,
                       n_jobs=4,
                       scoring="neg_root_mean_squared_error")

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

print("\n---------------MLPRegressor-------------------")

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

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

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = MLPRegressor()
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"MSE": make_scorer(mean_squared_error, greater_is_be
tter=False),
                            "MAE": make_scorer(mean_absolute_error, greater_is_b
etter=False)})

print("\nScore RMSE parâmetros default: ", ((-cv_results["test_MSE"]) ** (1 / 2
)).mean())
```

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

[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent wo
rkers.
[Parallel(n_jobs=4)]: Done  30 out of  30 | elapsed:   23.7s finishe
d

---------------MLPRegressor-------------------

Melhor conjunto de parâmetros:
 MLPRegressor(activation='relu', alpha=0.0001, batch_size='auto', be
ta_1=0.9,
             beta_2=0.999, early_stopping=False, epsilon=1e-08,
             hidden_layer_sizes=20, learning_rate='constant',
             learning_rate_init=0.001, max_fun=15000, max_iter=200,
             momentum=0.9, n_iter_no_change=10, nesterovs_momentum=T
rue,
             power_t=0.5, random_state=None, shuffle=True, solver='a
dam',
             tol=0.0001, validation_fraction=0.1, verbose=False,
             warm_start=False)

Melhor error score:
 1.8702426483245997

Score RMSE parâmetros default:  1.2823751962533572
```

## Árvore de decisão

Ao se utilizar uma única árvore de decisão com prunning variável (ccp_alpha é o hiperparâmetro sendo buscado), obtivemos 1.4868828491655717°C como RMSE do melhor ccp_alpha, sendo este de 0.007660778015155692. O resultado foi melhor do que o default do sklearn, que obteve RMSE de 1.5454842700795675°C com ccp_alpha de zero, ou seja, sem prunning.

In [10]:

```
# ============DecisionTreeRegressor=========================================

np.random.seed(1234)

# Gera os parametros de entrada aleatoriamente.
ccp_alpha = np.linspace(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 = DecisionTreeRegressor()
cv_results = \
    RandomizedSearchCV(estimator=regressor, cv=shuffle_splitter,
                       param_distributions=parametros,
                       verbose=1,
                       n_jobs=4,
                       scoring="neg_root_mean_squared_error")

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

print("\n--------------DecisionTreeRegressor------------------")

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

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

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = DecisionTreeRegressor()
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"MSE": make_scorer(mean_squared_error, greater_is_be
tter=False),
                            "MAE": make_scorer(mean_absolute_error, greater_is_b
etter=False)})

print("\nScore RMSE parâmetros default: ", ((-cv_results["test_MSE"]) ** (1 / 2
)).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:    4.0s finishe
d

--------------DecisionTreeRegressor------------------

Melhor conjunto de parâmetros:
 DecisionTreeRegressor(ccp_alpha=0.0044444444444444444, criterion='m
se',
                      max_depth=None, max_features=None, max_leaf_no
des=None,
                      min_impurity_decrease=0.0, min_impurity_split=
None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, presort='depreca
ted',
                      random_state=None, splitter='best')

Melhor error score:
 1.439673724344233

Score RMSE parâmetros default:  1.5409779988257635
```

## Random Forest

Utilizando o método Random Forest, que une múltiplas árvores de regressão por meio do voto majoritário, obtivemos um ganho elevado no RMSE, sendo de 0.9447367529767478°C com 1000 estimadores e no máximo 5 features, contra 1.0239742637984799°C 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, mas que aumentar a quantidade de features pode não ser uma boa opção quando se trabalha com múltiplas árvores (Random Forest).

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 atuano em conjunto, como é o caso.

In [11]:

```
# ============RandomForestRegressor=========================================

np.random.seed(1234)

# Gera os parametros de entrada aleatoriamente.
n_estimators = [10, 100, 1000]
max_features = [5, 10, 22]

# 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 = RandomForestRegressor()
cv_results = \
    GridSearchCV(estimator=regressor, cv=shuffle_splitter,
                 param_grid=parametros,
                 verbose=1,
                 n_jobs=1,
                 scoring="neg_root_mean_squared_error")

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

print("\n--------------RandomForestRegressor------------------")

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

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

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = RandomForestRegressor()
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"MSE": make_scorer(mean_squared_error, greater_is_be
tter=False),
                            "MAE": make_scorer(mean_absolute_error, greater_is_b
etter=False)})

print("\nScore RMSE parâmetros default: ", ((-cv_results["test_MSE"]) ** (1 / 2
)).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:  9.2min finishe
d

--------------RandomForestRegressor------------------

Melhor conjunto de parâmetros:
 RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='ms
e',
                      max_depth=None, max_features=5, max_leaf_nodes
=None,
                      max_samples=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=
0.0,
                      n_estimators=1000, n_jobs=None, oob_score=Fals
e,
                      random_state=None, verbose=0, warm_start=Fals
e)

Melhor error score:
 0.9447367529767478

Score RMSE parâmetros default:  1.0239742637984799
```

## GBM

Por último, trbalhamos com o GBM (Gradient Boosting Regressor), que adiciona modelos treinados diferentemente enfileirando a saída de alguns como entrada do seguinte, de forma que os sistemas sendo adicionados possam treinar sobre as regiões onde os modelos anteriores mais erraram.

O erro RMSE obtido com os melhores parâmetos encontrados foi de 1.0735283782351117°C, sendo os parâmetros de: n_estimators = 96, learning_rate = 0.3 e max_depth = 3. Pode-se concluir que mais estimadores tornam o modelo melhor, o learning rate de 0.3 simplesmente era pequeno o suficiente para a atividade e uma profundidade maior na árvore também ajuda a classficação.O RMSE obtido com o default foi de 1.2233210699332346°C.

O RMSE deste método se aproximou do Random Forest e do SVR-RBF, que foram os melhores do estudo, indicando que se não for preciso um RMSE extremamente baixo, o tempo de treinamento deste modelo pode compensar, já que os métodos supracitados foram ordens de grandeza mais lentos que este.

In [12]:

```
# ============GradientBoostingRegressor=====================================

np.random.seed(1234)

# Gera os parametros de entrada aleatoriamente.
n_estimators = np.linspace(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 = GradientBoostingRegressor()
cv_results = \
    RandomizedSearchCV(estimator=regressor, cv=shuffle_splitter,
                       param_distributions=parametros,
                       verbose=1,
                       n_jobs=4,
                       scoring="neg_root_mean_squared_error")

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

print("\n--------------GradientBoostingRegressor------------------")

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

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

shuffle_splitter = ShuffleSplit(n_splits=5, test_size=0.3, random_state=1234)
regressor = GradientBoostingRegressor()
cv_results = \
    cross_validate(estimator=regressor, X=X_data_scaled, y=y_data,
                   cv=shuffle_splitter,
                   scoring={"MSE": make_scorer(mean_squared_error, greater_is_be
tter=False),
                            "MAE": make_scorer(mean_absolute_error, greater_is_b
etter=False)})

print("\nScore RMSE parâmetros default: ", ((-cv_results["test_MSE"]) ** (1 / 2
)).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  42 tasks      | elapsed:   16.2s
[Parallel(n_jobs=4)]: Done  50 out of  50 | elapsed:   17.5s finishe
d

--------------GradientBoostingRegressor-----------------

Melhor conjunto de parâmetros:
 GradientBoostingRegressor(alpha=0.9, ccp_alpha=0.0, criterion='frie
dman_mse',
                           init=None, learning_rate=0.3, loss='ls', m
ax_depth=3,
                           max_features=None, max_leaf_nodes=None,
                           min_impurity_decrease=0.0, min_impurity_sp
lit=None,
                           min_samples_leaf=1, min_samples_split=2,
                           min_weight_fraction_leaf=0.0, n_estimators
=100,
                           n_iter_no_change=None, presort='deprecate
d',
                           random_state=None, subsample=1.0, tol=0.00
01,
                           validation_fraction=0.1, verbose=0, warm_s
tart=False)

Melhor error score:
 1.067704037583362

Score RMSE parâmetros default:  1.2233141636172036
```

## Conclusão

Os métodos de Random Forest e SVR com RBF retornaram os melhores resultados em termos de RMSE, tendo porém um tempo de execução muito elevado. O GBM foi o terceiro melhor regressor e com desempenho próximo a estes 2, tendo um tempo de execução muito menor, sendo da ordem de segundos contra minutos ou horas dos primeiros, sendo preferível utilizá-lo quando tempo de treinamento for uma métrica crítica.

In [ ]:

```
# Shutdown PC at the end of training
import os
import time

time.sleep(10 * 60)
os.system("shutdown 0")
```